RUNTIME-ADAPTIVE GENERALIZED TASK PARALLELISM

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of Saarland University

by

Kevin Streit

Saarbrücken, 2017

UNIVERSITÄT
DES
SAARLANDES

Day of Colloquium	20 / 10 / 17
Dean of the Faculty	Univ.-Prof. Dr. Frank-Olaf Schreyer
Chair of the Committee	Prof. Dr. Jan Reineke
Reporters	
First reviewer	Prof. Dr. Andreas Zeller
Second reviewer	Prof. Dr. Sebastian Hack
Third reviewer	Prof. Christian Lengauer, Ph.D.
Academic Assistant	Dr. María Gómez Lacruz

To my wife Lena and my little ladies Lotta and Ida, who always showed the patience and understanding to support me in writing this thesis.

SAARLAND UNIVERSITY

Zusammenfassung

Chair of Software Engineering and Compiler Design Lab
Department of Computer Science — Saarland Informatics Campus

by Kevin Streit

Mehrkernsysteme sind heutzutage allgegenwärtig und finden täglich weitere Verbreitung. Und während, limitiert durch die Grenzen des physikalisch Machbaren, die Rechenkraft der einzelnen Kerne bereits seit Jahren stagniert oder gar sinkt, existiert bis heute keine zufriedenstellende Lösung zur effektiven Ausnutzung der gebotenen Rechenkraft, die mit der steigenden Anzahl an Kernen einhergeht.

Existierende Ansätze der automatischen Parallelisierung sind häufig hoch spezialisiert auf die Ausnutzung bestimmter Programm-Muster, und somit auf die Parallelisierung weniger Programmteile. Hinzu kommt, dass häufig verwendete invasive Laufzeitsysteme die Kombination mehrerer Parallelisierungs-Ansätze verhindern, was der Praxistauglichkeit und Reichweite automatischer Ansätze im Wege steht.

In der Ihnen vorliegenden Arbeit zeigen wir, dass die Spezialisierung auf eng definierte Programmuster nicht notwendig ist, um Parallelität in Programmen verschiedener Domänen effizient auszunutzen. Wir entwickeln einen generalisierenden Ansatz der Parallelisierung, der, getrieben von einem mathematischen Optimierungsproblem, in der Lage ist, fundierte Parallelisierungsentscheidungen unter Berücksichtigung relevanter Kosten zu treffen. In Kombination mit einem spezialisierenden und adaptiven Laufzeitsystem ist der entwickelte Ansatz in der Lage, mit den Ergebnissen spezialisierter Ansätze mitzuhalten, oder diese gar zu übertreffen.

Diese Arbeit ist in englischer Sprache verfasst.

SAARLAND UNIVERSITY

Abstract

Chair of Software Engineering and Compiler Design Lab
Department of Computer Science — Saarland Informatics Campus

by Kevin Streit

Multi core systems are ubiquitous nowadays and their number is ever increasing. And while, limited by physical constraints, the computational power of the individual cores has been stagnating or even declining for years, a solution to effectively utilize the computational power that comes with the additional cores is yet to be found.

Existing approaches to automatic parallelization are often highly specialized to exploit the parallelism of specific program patterns, and thus to parallelize a small subset of programs only. In addition, frequently used invasive runtime systems prohibit the combination of different approaches, which impedes the practicality of automatic parallelization.

In the following thesis, we show that specializing to narrowly defined program patterns is not necessary to efficiently parallelize applications coming from different domains. We develop a generalizing approach to parallelization, which, driven by an underlying mathematical optimization problem, is able to make qualified parallelization decisions taking into account the involved runtime overhead. In combination with a specializing, adaptive runtime system the approach is able to match and even exceed the performance results achieved by specialized approaches.

Acknowledgements

First and foremost, I want to particularly thank my advisers Andreas and Sebastian for their time, patience and advice. They never stopped to believe in me and a risky research idea and provided support and inspiration throughout the years.

While I had the pleasure to meet many interesting and highly valued people that have influenced and contributed to the work that has been condensed in this very PhD thesis by endless and fruitful discussions, I have to particularly mention two of my colleagues with whom I closely collaborated and whom I consider my friends:

The implementation and ideas of the whole *Sambamba* framework, which is an integral part of the work presented in this thesis, has been done together with Clemens Hammacher, whose main topic of research lies in the field of speculative parallelization. He has been colleague, teammate and a good friend during the whole course of my thesis work. Consequently, not only the implementation, but also many ideas and solutions presented on the following pages are in the end the result of our collaboration.

In no way fewer influence on the presented results had Johannes Doerfert, a former Bachelor and Master student of ours who now conducts his PhD studies in the field of Program Optimizations based on the Polyhedral Model in the Compiler Design Lab at Saarland University. He also contributed in the form of code, time and discussions to many parts of the presented work.

Part of the work presented in this thesis was performed in the context of the Software-Cluster project *EMERGENT* (http://www.software-cluster.org). It was funded by the German Federal Ministry of Education and Research (BMBF) under grant no. “01IC10S01”. Later work has been supported, also by the German Federal Ministry of Education and Research (BMBF), through funding for the Center for IT-Security, Privacy and Accountability (CISPA) under grant no. “16KIS0344”.

Contents

Chapter 1

Introduction and Motivation

The free lunch is over. This frequently cited phrase has been coined by Herb Sutter in his famous article [1] proposing a necessary reinterpretation of the consequences of *Moore's Law* [2]. Gordon E. Moore predicted in 1965 roughly a biennial doubling of the number transistors in dense integrated circuits. Surprisingly, this prediction, which was meant to cover the next ten years, held true for several decades and still does. The consequences changed, however.

Consider Figure 1.1, which plots the development of transistor counts in Intel processors from 1970 to 2010 (note the log-scale of the y-axis). For many years, processor manufacturers have been able to translate the increased number of transistors directly into higher clock speeds and increased instruction-level parallelism (ILP); this however changed dramatically around 2004 when those numbers started to stagnate due to increasing difficulties with dissipating the additionally produced heat that comes with an increasing power consumption.

Unfortunately, the increasing clock-speed and exploitable instruction-level parallelism have been the driving forces of increasing single-thread performance from which most programs have been able to effortlessly profit—the free lunch. As can be seen in Figure 1.2, the additional transistors, whose number still grows as predicted by Moore, are put into an exponentially growing number of compute cores.

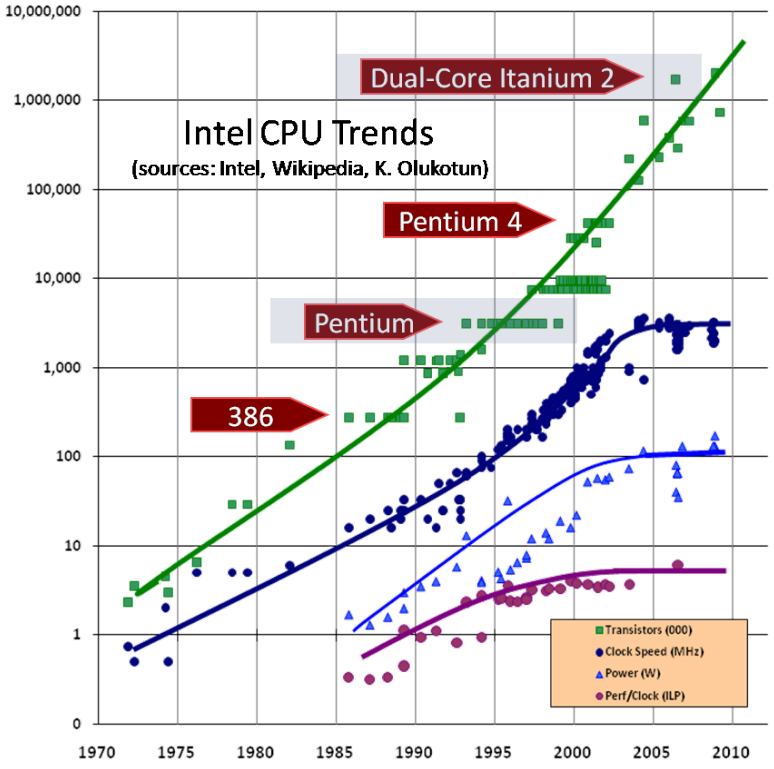


FIGURE 1.1: Development of the number of transistors 1970-2010. Source: Sutter [1].

Now to exploit the power of modern multi-core architectures, the software being executed on it has to run in parallel. While there are multiple incarnations of parallel execution, like running multiple applications at the same time or running multiple instances of a program or heavy computation in parallel, developers inevitably have to face the problem of also *internally* parallelizing a specific application. This also applies to legacy applications, i.e., applications which are typically not under active development, and, consequently, the original developer not necessarily available any more. Unfortunately, parallelizing software and maintaining the parallel code is expensive, error-prone and generally considered hard by most programmers of which many say about themselves that they are either unable or unwilling to explicitly deal with parallelization [4, 5]. These difficulties particularly apply to legacy software, which is oftentimes hard enough to understand and maintain in a sequential form, let alone in a platform-specific, parallelized form.

Consider the *kernel_bicg* function shown in Figure 1.3. It is an *OpenMP* [6, 7] parallelized version of the code shown in Figure 1.4a, as produced by a state-of-the art polyhedral

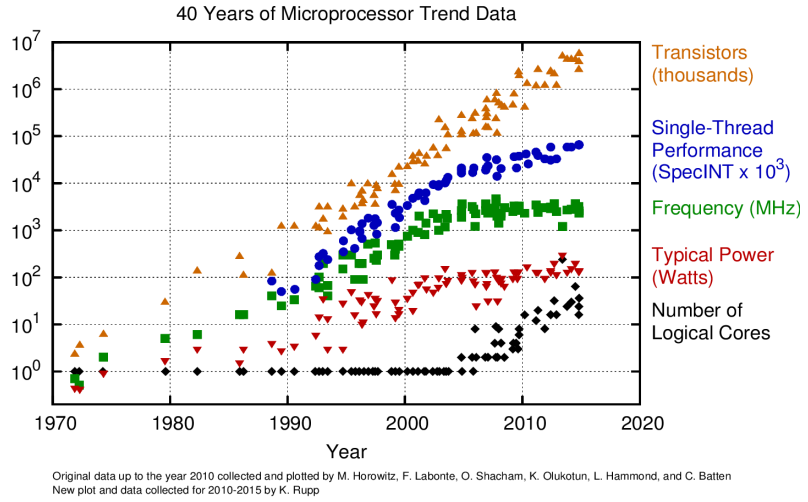


FIGURE 1.2: Development of transistor count, single-core performance, and number of cores 1970-2015. Source: Rupp [3].

optimization tool. Performance-wise this version is desirable; but due to its complexity it is not the code developers should need to maintain. Instead it should be the easily reproducible result of the press of a button in the course of compiling for a new platform or changing the underlying algorithm as shown in Figure 1.4a.

Due to the increased complexity of producing and maintaining parallel code, automatic parallelization of sequential code has been a central goal of academic computing research and the past decades have produced several concepts and parallelization approaches. Each one is tailored to exploit parallelism found in *specific program patterns*: Given a suitable reduction analysis, any *DOALL-style loop parallelizer* [8] can enable parallel execution of each of the three loops in Figure 1.4a. Loops which carry dependences do not qualify for *DOALL*-style parallelization. They can be dealt with by employing *DOACROSS-style loop parallelization* [9] or *Software Pipelining*[1], for instance. The manually tail-call optimized version of *quicksort* in Figure 1.4c, is an example for such a dependency-carrying loop. Recursive tasks, as shown in Figure 1.4b, can be parallelized by classical fork-join style *task parallelism*.

The examples mentioned above are a first hint on the diverse appearance of parallelism on a very high level of abstraction. In fact, the patterns searched for and exploited by parallelization approaches discussed in modern parallelization research are way more

[1] sometimes also called *DOPIPE*

```
1  static void kernel_bicg(...) {
2   if ((ny >= 1)) {
3    ub1 = floord((nx + -1), 256);
4  #pragma omp parallel for private(c2, c3, c4, c5, c6) firstprivate(ub1)
5    for (c1 = 0; c1 <= ub1; c1++)
6      for (c2 = 0; c2 <= floord((ny + -1), 256); c2++)
7        for (c3 = (8 * c1); c3 <= min(floord((nx + -1), 32), ((8 * c1) + 7)); c3++)
8          for (c4 = (8 * c2); c4 <= min(floord((ny + -1), 32), ((8 * c2) + 7)); c4++)
9            for (c5 = (32 * c4); c5 <= min(((32 * c4) + 31), (ny + -1)); c5++)
10 #pragma ivdep
11 #pragma vector always
12 #pragma simd
13               for (c6 = (32 * c3); c6 <= min(((32 * c3) + 31), (nx + -1)); c6++)
14                 q[c6]=q[c6]+A[c6][c5]*p[c5];
15   }
16   if ((nx >= 1)) {
17     ub1 = floord((ny + -1), 256);
18 #pragma omp parallel for private(c2, c3, c4, c5, c6) firstprivate(ub1)
19     for (c1 = 0; c1 <= ub1; c1++)
20       for (c2 = 0; c2 <= floord((nx + -1), 256); c2++)
21         for (c3 = (8 * c1); c3 <= min(floord((ny + -1), 32), ((8 * c1) + 7)); c3++)
22           for (c4 = (8 * c2); c4 <= min(floord((nx + -1), 32), ((8 * c2) + 7)); c4++)
23             for (c5 = (32 * c4); c5 <= min(((32 * c4) + 31), (nx + -1)); c5++)
24 #pragma ivdep
25 #pragma vector always
26 #pragma simd
27               for (c6 = (32 * c3); c6 <= min(((32 * c3) + 31), (ny + -1)); c6++)
28                 s[c6]=s[c6]+r[c5]*A[c5][c6];
29   }
30 }
```

FIGURE 1.3: An example of complex, but efficient parallelization: parallel *BiCG*.

specific. The following quotation has been taken from a recent publication [10] on so-called speculative cross-invocation parallelization of nested loops:

> A hot loop is a candidate for SpecCross if it satisfies three conditions:
>
> 1. the outermost loop itself cannot be successfully parallelized by any automatic parallelization technique implemented in the existing parallelizing compiler infrastructure (including DOALL, LOCALWRITE, and DSWP);
> 2. each inner loop can be independently parallelized by a non-speculative and non-partition based parallelization technique such as DOALL and LOCALWRITE; and
> 3. the sequential code between two inner loops can be privatized and duplicated among all worker threads.

Searching for and exploiting such specific patterns raises the question of applicability to a broader range of applications. This is not to say that every parallelization approach should be able to deal with every form of exploitable parallelism or even every application domain. However, a big challenge of parallelization nowadays are general purpose applications.

In contrast to typically well understood and statically analyzable scientific applications, such general purpose applications show irregular and input-dependent memory accesses and dependence patterns and are effectively assembled from a whole set of dynamically linked and at compile-time unavailable libraries. But most importantly, they are typically implemented by a diverse set of differently skilled and trained programmers making use of completely diverse computational patterns in different parts of an application. In contrast to a typical scientific application, one cannot easily identify the one deep loop nest that implements the computational kernel and dominates execution time.

All the parallelization approaches mentioned before work well for the specific form of parallelism they have been developed for; however, they are mostly also *restricted* to

```
1  void kernel_bicg(double A[NX][NY], double s[NY], double q[NX],
2                   double p[NY], double r[NX]) {
3
4    for (int i = 0; i < NY; i++)
5      s[i] = 0;
6
7    for (int i = 0; i < NX; i++) {
8      q[i] = 0;
9
10     for (int j = 0; j < NY; j++) {
11       s[j] = s[j] + r[i] * A[i][j];
12       q[i] = q[i] + A[i][j] * p[j];
13     }
14   }
15 }
```

(A) *BiCG*

```
1  void fft_twiddle(int i, int i1, COMPLEX *in, COMPLEX *out, COMPLEX *W,
2                   int nW, int nWdn, int r, int m) {
3
4    if (i == i1 - 1) {
5      fft_twiddle_gen1(in + i, out + i, W, r, m, nW, nWdn * i, nWdn * m);
6    } else {
7      int i2 = (i + i1) / 2;
8      fft_twiddle(i,i2,in,out,W,nW,nWdn,r,m);
9      fft_twiddle(i2,i1,in,out,W,nW,nWdn,r,m);
10   }
11 }
```

(B) *fft*

```
1  void seqquick(ELM * low, ELM * high) {
2    ELM *p;
3    while (high - low >= 1) {
4      p = seqpart(low, high);
5      seqquick(low, p);
6      low = p + 1;
7    }
8  }
```

(C) *Quicksort*

FIGURE 1.4: Different parallelizable functions: *BiCG* (A) kernel implementation (taken from the PolyBench 3.2 suite); and *fft* (B) and *Quicksort* (C) implementations (adapted from the *cilksort* and *fft* programs of the *Cilk* example application suite).

be effective only on these patterns they seek to exploit. We are not aware of a single approach that efficiently parallelizes all of the three code examples in Figure 1.4, let alone handling code which embodies complex combinations of loops and recursion across several functions. Indeed, an integration of these parallelization approaches poses tremendous practical challenges, as the underlying models and assumptions are vastly different. This in particular holds true if the parallelization approach relies on a complex runtime system which is typically designed under the assumption of sole control over the computational

resources of a system. Different parallelized applications running in parallel, or even differently parallelized parts of a single application violate this assumption.

But even if we could build a compiler that provides all important parallelization approaches and covers any specific form of parallelism, we still would lack a joint cost model that is powerful enough to drive the choice between different kinds of parallelism: if a compiler has to decide for one approach to parallelize a given piece of code, how should it decide? In particular, being aware of the fact that it should decide for only one system used for the whole application, as the runtime systems of different approaches are typically not compatible or at least interfere in unpredictable ways, undermining the fragile cost models driven by heuristics and platform-specific benchmarks. Furthermore, as the benefits of parallelization outside of the scientific domain are still hardly predictable, most approaches rely on the option to fall back to sequential execution if the runtime system deems parallel execution unbeneficial. But if the one system chosen by the compiler at compile-time falls back to sequential execution at runtime, the chance for different parallelization approaches to prove successful is gone.

An integrated approach, however, would be worthwhile to combine the strengths of the present approaches for parallelization. The need for integration becomes even more obvious when *speculation*, *privatization*, and *reduction* are taken into account—three techniques that have been identified frequently as being among the most important techniques for enabling parallelism. Scientists even go as far as claiming that without such techniques efficient parallelization of general purpose applications is impossible (e.g., [11] or [12]). Enabling parallelism naturally increases the range of possibilities for a parallelizer to choose from. However, it is not automatically implied that such parallelism can be exploited in a profitable way. Each of those techniques introduces potentially significant overhead. This overhead needs to be compensated by an at least equally high reduction of execution time.

As a detailed example, reconsider *BiCG* in Figure 1.4a. Even without considering complex iteration space transformations as necessary to produce the result shown in Figure 1.3, we are left with multiple forms of parallel execution to choose from: different opportunities exist to realize a reduction, for instance, and an automatic parallelizer is left with the decision of which loop to parallelize. When parallelizing the outer loop of the loop-nest,

the reduction-induced dependence via the array s needs to be broken and handled specially in the parallel code. This can be done by privatizing the whole array s, techniques like *LOCALWRITE* [13], or, for instance, by using atomic operations (or unordered atomic sections) instead. An alternative would be to parallelize the innermost loop only. In that case, only one array cell needs to be privatized in order to fix the broken reduction dependence via the array q. Making a qualified decision on the way to parallelize the code and fix the broken reduction dependences, or even to speculate on statically unanalyzable dependences requires to take multiple factors into account, some of which are not known at compile-time: user input, execution platform, available number of cores. All this calls for a *deep integration of parallelization approaches on a sound theoretical base, implemented on a stable and powerful platform for compile-time and runtime analysis.*

1.1 Contributions of this Thesis

The contribution of this thesis is a unified and generalizing approach of parallelization including the prototypical implementation of *Sambamba*, a hybrid compile-time/runtime parallelization framework based on the *LLVM*-compiler infrastructure. At the core of the thesis, we introduce the concept and implementation of *generalized task parallelism*—a single unified framework for automated parallelization. The main contributions of the thesis are as follows:

Static/dynamic Compiler Framework for Program Parallelization We provide the prototypical implementation of our *Sambamba* framework based on the *LLVM* compiler infrastructure. *Sambamba*, which is made available as open-source, can be used as the technical foundation of further research in hybrid static/dynamic parallelization approaches.

Uniform Program Representation We present a uniform program representation based on the program dependence graph (*PDG*) (Section 5.1). Relevant properties like profiling information, reduction, privatizability and speculation opportunities are broken down to the dependence level and correspondingly represented in the PDG.

Parallelization as Optimization Problem Based on this representation, we *reduce the problem of parallelization to linear optimization to find local parallelization candidates* (Chapter 6). The formulation is independent of any special form of parallelism, and integrates central aspects of existing approaches without reimplementing them. Optimization is driven by a *cost model* derived from static profile estimates and runtime profiling information. Implementations of this approach do not rely on special code features. Loop structures, for instance, are completely transparent: existing loops can be fully (i.e., DOALL-style) or partially (in case of carried dependences) parallelized, while still allowing for loop-independent task parallelism.

Specializing Parallel Code Generation We show how to effectively generate *specialized parallel code* (or, simply speaking: optimized parallel code) from the found generalized parallelism. Together with an adaptive runtime system that can continuously reassess parallelization decisions, our parallelizer $ParA_{\gamma}$ is able to match the performance of specialized parallelization approaches (Chapter 7).

Evaluation We *evaluate* the individual parts of our system (Chapter 9) on a set of programs from various benchmarks suites, showing that generalized task parallelism

1. subsumes and integrates different and independent forms of parallelism;
2. discovers parallelization opportunities similar to those found by experts; and
3. produces efficient parallel code for a broad range of applications.

Semi-automatic Parallelization Finally, we provide the implementation of a semi-automatic parallelization toolchain based on *Sambamba* and integrated into the *Eclipse IDE*, which is able to not only parallelize parts of the application under development automatically, but also to verify, improve and finally implement parallelization hints and decisions given by the developer in the form of very simple, *OpenMP*-style program annotations.

1.2 Publications

This thesis builds on the following publications, listed in descending chronological order.

Thread-Level Speculation with Kernel Support [14]. Clemens Hammacher, Kevin Streit, Andreas Zeller, and Sebastian Hack. In Proceedings of the 25th International Conference on Compiler Construction (CC), March 2016.

Generalized Task Parallelism [15]. Kevin Streit, Johannes Doerfert, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. In ACM Transactions on Architecture and Code Optimization (TACO), Volume 12, Number 1, April 2015.

Polly's Polyhedral Scheduling in the Presence of Reductions [16]. Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. 5th International Workshop on Polyhedral Compilation Techniques (IMPACT), January 2015.

Sambamba: Runtime Adaptive Parallel Execution [17]. Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. In Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing Systems (ADAPT), January 2013.

SPolly: Speculative Optimizations in the Polyhedral Model [18]. Johannes Doerfert, Clemens Hammacher, Kevin Streit, and Sebastian Hack. 3rd International Workshop on Polyhedral Compilation Techniques (IMPACT), January 2013.

Sambamba: A Runtime System for Online Adaptive Parallelization [19]. Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. In Proceedings of the 21st International Conference on Compiler Construction (CC), March 2012.

Profiling Java Programs for Parallelism [20]. Clemens Hammacher, Kevin Streit, Sebastian Hack, and Andreas Zeller. In Proceedings of the ICSE Workshop on Multicore Software Engineering (IWMSE), May 2009.

Chapter 2

Background Terminology and Concepts

The purpose of this chapter is to introduce important background terminology and concepts used but not defined within this thesis. Also terminology is added which will be used throughout this thesis, potentially before it is fully defined. A reader with a background in compiler construction or parallelization may safely skip this section and use it as a glossary to return and read about unknown or unclear terminology only. We will keep definitions on an intuitive level and refer the interested reader to the respective literature where appropriate.

ϕ-nodes, or ϕ-instructions are virtual instructions selecting among a given set of values based on their *dynamic control flow* predecessor. They are an essential part of *SSA*-based compiler *IR*s allowing to make *def-use relations* explicit and fulfill the single assignment requirement. ϕ-nodes are virtual in the sense that they usually do not have a direct correspondence in the finally produced machine code. [see Cytron et al. [21] for details on SSA and ϕ-nodes].

Atomic section is a section of code which is executed atomically, i.e., its effects, like memory effects, are either completely visible to outside observers or not at all. No intermediate state or subset of effects will be visible at any given point in time.

Atomic sections play a major role in *speculation* systems, *reduction* realization, or generally in *concurrent* execution as a mechanism of synchronization.

Automatic parallelization refers to the process of *parallelization* without involving the developer. It is typically performed *statically* by an automatically parallelizing compiler, or *dynamically* by a parallelizing runtime system. Sections 3.3 and 3.4 give an overview of research on automatic parallelization relevant in the context of this thesis.

Basic block is a linear sequence of instructions. The instructions contained in one basic block are all executed (if the block is entered) or not at all (if the block is not entered). Basic blocks typically form the vertices of a *control flow graph*. [see Allen [22] for more details].

Call instruction (LLVM) is an *LLVM* instruction calling a function. Call instructions (and *invoke instructions*) are interesting *parallelization* targets as they typically represent an isolated and encapsulated piece of work (the *callee*). [see `http://llvm.org/docs/doxygen/html/classllvm_1_1CallInst.html#details` for more details].

Callee refers to the function called through a *call* or *invoke* instruction.

Caller refers to the function containing a given *call* or *invoke* instruction.

Common subexpression elimination (CSE) is a compiler optimization seeking to minimize repeated computation of the same expression on any given *control flow path*. The computation of the subject expression is placed at a program point *dominating* multiple appearances, which in turn are replaced by a usage of the new definition.

Concurrency is the concept of executing multiple processes (or arbitrary pieces of code) at the same time. Note that this does not necessarily imply that at any given point in time, multiple processes are executing simultaneously. Concurrency might as well be implemented by time slicing, i.e., switching between execution of multiple started but not yet finished processes. *Parallelism* is another form of concurrency.

Conflict in the context of this thesis refers to the interference of two or more instructions or pieces of code which execute *concurrently*. Conflicts can for instance be caused by

concurrent and unprotected accesses to the same memory region involving at least one write access. Equations 4.1 to 4.6 on page 45 formally define a *memory conflict* in our setting.

Control dependence refers to the execution of an instruction (or *basic block*) B being dependent on a branching decision made by another instruction A. We say A contributes to the decision to execute B or not, or B is control-dependent on A. [see Kennedy and Allen [8] for more details].

Control flow graph (CFG) is a graph representation of a program. *Basic blocks* typically form the vertices of the CFG with edges representing transfer of execution. Paths through the CFG represent a static overapproximation of possible execution paths. Typically, a program is decomposed into functions with each function having its own CFG. [see Allen [22] for more details].

Control flow path is a path through the *control flow graph* representing a potentially possible path of execution. A control flow path is a *static* construct. It is not guaranteed that a program input exists which *dynamically* triggers execution of this path. [see Allen [22] for more details].

Critical path in the context of parallel execution refers to the longest path of sequentially executing processes or tasks. The length of the critical path determines the overall execution time of a parallel program. The goal of parallelization typically is to minimize the critical path execution time.

Cross invocation parallelism refers to the parallelization of subsequent *dynamic* invocations of a loop. This is in contrast to classical approaches which parallelize a single invocation of a loop, followed by explicit synchronization before proceeding with the next invocation.

Data dependence is a dependence between two computational units (e.g., instructions, *basic blocks*, *PDG* subgraphs, ...), in which one unit depends on the data produced by another unit. In this thesis, we say B depends on A, denoted by $B \rightarrow A$[1].

[1] In the literature you will also find the arrow going in the other direction, i.e., $A \rightarrow B$ to state that B depends on A. This denotation resembles the direction of the control flow or the data, while we decided to use a notation expressing the dependence itself.

Subsection 4.1.1 formally defines our notion of dependence, while Kennedy and Allen [8] provide further background on the classical notion of dependence.

Data structure analysis (DSA) refers to a scalable, context-sensitive and flow-insensitive points-to-analysis designed and implemented for the *LLVM* compiler infrastructure. *DSA* is used in this thesis as the foundation of the *data dependence* analysis. Subsection 4.1.1 provides a detailed description of *DSA* and its most important properties while the full description can be found in Lattner et al. [23].

Decoupled software pipelining (DSWP) is an approach to loop parallelization by decomposing the loop into multiple pipeline stages executing in parallel. The idea in short is to split the loop body into stages which can be ordered in such a way that dependences only go backward in the list of stages, loop-carried dependences are only allowed within a stage. Each stage then forms a separate loop executing in parallel to all other stages while dependences are fulfilled be explicit synchronization and communication of produced values from the producing to the consuming stage. We shortly describe multiple approaches from the *DSWP* family on page 29.

def-use relation or *data-flow* is a relation between instructions, denoted $s \overset{\circ}{\rightarrow} t$ in this thesis, with the meaning of s defining a value that is used by t. The def-use relation is used in this thesis to define reduction properties on page 77.

DOACROSS loop is, like *DOALL*, one of the classical loop parallelization techniques. It refers to a parallelized loop having loop-carried dependences. Correctness is guaranteed by introducing explicit synchronization to delay execution of the source of a *data dependence* until the target is ready. [see Cytron [9] for more details].

DOALL loop is the classical parallel loop having no *loop-carried dependences*. All iterations of the loop can be executed in parallel to each other or in an arbitrary order. [see Kennedy and Allen [8] for more details].

Dominance is a relation over nodes of the *control flow graph*. A node A is said to *dominate* (or *pre-dominate*) node B if A is contained in every *control flow path* starting from the dedicated entry node of the CFG and ending in B. [see Allen [22] for more details].

Dynamic in the context of this thesis is used as a synonym for "at runtime", i.e., while an application is running. This is in contrast to *static* meaning without requiring the program to be executed, typically *at compile-time*. Dynamic program analysis collects and interprets information during the (possibly artificial) execution of the target program. Static analysis in contrast deduces information solely from a representation of the program itself.

General purpose applications are large applications fulfilling multiple purposes by providing *dynamically* triggered functionality. They are typically complex, make use of a diverse set of *irregular data structures* and show *statically* hard or impossible to predict control-flow and memory access patterns (*irregular application*). Due to these properties general purpose applications, in contrast to mathematical or scientific applications, pose a grand challenge to *automatic parallelization*.

Granularity (parallelization) refers to the size of the *dynamic* computational units a parallelizable program (part) is decomposed into. In the context of this thesis we refer to such a computational unit as *task*. If parallel tasks are too small (fine-grained), then the overhead of packing and spawning such a task outweighs the actual work done in the task, and the dynamic task scheduler gets *oversubscribed*. If the tasks are too coarsely defined, i.e., too large, then parallelism suffers and the task scheduler has not enough freedom to balance parallel execution. Efficient and profitable *automatic parallelization* needs to find the right trade-off between parallelism and overhead.

Integrated development environment (IDE) refers to an application integrating multiple tools important to software development. Such tools typically include a source code editor, compiler, debugger, source code management, and several analyses to point the developer to possible errors and flaws. Chapter 10 describes the integration of our parallelization tools into the Eclipse IDE (`http://www.eclipse.org`).

Invoke instruction (LLVM) is an *LLVM* instruction invoking a function which might throw an exception. In contrast to *call instructions*, an *invoke* has two control flow successors: one representing regular control flow, and one representing the exceptional control flow. *Invoke* instructions (and *call instructions*) are interesting *parallelization* targets as they typically represent an isolated and encapsulated piece

of work (the *callee*). [see `http://llvm.org/docs/doxygen/html/classllvm_1_1InvokeInst.html#details` for more details].

Intermediate representation (IR) refers to a program representation used internally by the compiler. A program is typically translated into multiple IRs during compilation from source code to machine code. The *program dependence graph* and the parallel control flow graph (see Subsection 4.1.3) are the main IRs used in this thesis; details on the main IR used by the *LLVM* compiler infrastructure can be found at `http://llvm.org/docs/LangRef.html`.

Irregular applications are applications using *irregular data structures* and/or employing irregular, i.e., conditional and possibly input-dependent, control flow and *memory access patterns*. Due to their *statically* unpredictable behavior and properties, irregular applications pose a grand challenge to automatic parallelization. [see Yelick [24] for more details].

Irregular data structures are typically pointer-based data structures composed from *dynamically* allocated, i.e., non-contiguous, memory connected via pointers. Irregular data structures include graphs, hashmaps, and unbalanced trees for instance. The behavior (memory accesses or execution time) of algorithms working with or traversing irregular data structures is oftentimes input-dependent and hard or impossible to statically predict, which poses challenges to effective *automatic parallelization*. [see Yelick [24] for more details].

Irregular memory access patterns are *statically* unpredictable memory access patterns typically induced by traversing *irregular data structures*. This is in contrast to regular memory access patterns which arise, for instance, from traversing an array. [see Yelick [24] for more details].

Irregular dependence pattern means a *statically* unpredictable dependence pattern, typically of *loop-carried dependences*, whose existence and dependence distance are not statically determined. Irregular dependence patterns pose a challenge to automatic parallelization, for instance when efficiently placing synchronization primitives.

ILP (Instruction-level parallelism) in this thesis refers to hardware-exploited instruction-level parallelism as used by modern CPUs through out-of-order execution and pipelining. Such a CPU is able to execute multiple instructions which are close-by on a sequential instruction stream in parallel to each other. Instruction-level parallelism is typically exploited by the hardware or the compiler without any interaction by the developer.

ILP (Integer linear programming), or integer programming, is a special form of *linear optimization* in which some or all variables are restricted to be integer-valued. In contrast to real-valued linear optimization, which is known to be solvable in polynomial time, integer programming is provably NP-hard.

Inter-procedural Analysis/Optimization is an analysis/optimization technique propagating analysis results through *call* or *invoke* instructions, i.e., from *callers* to *callees* or vice versa. Inter-procedural analyses are typically more powerful than *intra-procedural analyses* but also more expensive.

Intra-procedural Analysis/Optimization in contrast to *inter-procedural analysis* is limited to working on a single function. *Call* or *invoke* instructions are typically treated conservatively, in the worst case assuming anything can happen when calling another function.

Linear optimization, or linear programming, is a mathematical optimization method. The goal of linear optimization is to minimize or maximize a linear objective function subject to linear inequalities by choosing values for a set of real-valued variables.

LLVM is a modern, *SSA*-based compiler infrastructure used as a backend by many modern programming languages and as the basis of many program analyses and transformations. LLVM is also used as the technical basis of this thesis work. [see `http://www.llvm.org` for more details].

LOCALWRITE is a loop parallelization technique based on the owner-computes rule. In case of a loop writing to a *regular data structure*, possibly inducing loop-carried dependences, *LOCALWRITE* attempts to partition the iteration space in such a way that parallel threads executing disjoint parts of the iteration space also write to

disjoint parts of the target array. If this is statically impossible, for instance due to the target data structure being indirectly addressed, the iteration space is not completely partitioned, but (partially) replicated among threads. Before writing to the target data structure, each thread checks the target index is part of its assigned range, possibly throwing away computed values if this is not the case. [details can be found in Han and Tseng [13]].

Loop nest refers to multiple nested loops, i.e., loops contained in another.

Loop-carried dependence, or cross-iteration dependence, refers to a *data dependence* that spans multiple iterations of the loop. I.e., the source and the target of the dependence are not contained in the same *dynamic* loop iteration.

Memory effect refers to the possibly externally observable effect caused by writing to (write effect) or reading from (read effect) memory.

Non-memory effect refers to any non-memory-induced externally observable effect caused by executing a program statement. Non-memory effects in the context of this thesis include *termination effects*, which terminate the execution of the function under observation, non-memory read effects (like for instance reading the system clock), and non-memory write effects (like printing to the screen).

Online adaptive optimization refers to a *dynamic* program optimization being performed at runtime, taking into account information collected during the ongoing execution. Online adaptive optimization typically includes *just-in-time compilation* capabilities to compile and use the online optimized program parts.

OpenMP (Open Multi-Processing) is a wide-spread scalable and platform-independent API for shared memory multiprocessing in C, C++, and FORTRAN. It consists of a set of compiler directives and libraries used by the programmer to explicitly describe and exploit the parallelism of an application. [see Dagum and Menon [6] and [7] for more details].

Oversubscription in the context of this thesis refers to the inability of the computational facilities involved in the execution of parallel tasks (dynamic scheduler and compute-cores for instance) to process the tasks at least as fast as they are produced. In case

of oversubscription there is no use in further spawning (i.e., producing) parallel tasks further increasing the pressure on an already overloaded system.

Parallelism is a special form of *concurrency* in which at any given point in time multiple concurrently running tasks might be simultaneously executing on different computational facilities, like for instance multiple cores of a CPU.

Parallelization is the process of transforming a sequential application to a parallel application, typically with the goal to speed-up execution.

***ParA*$_\gamma$** refers to the generalized task parallelization scheme designed and developed in this thesis. Chapters 5 to 8 describe different aspects of $ParA_\gamma$ in thorough detail.

***ParA*$_\tau$** refers to the simple intra-block *parallelization* scheme developed on top of *Sambamba*. Section 4.1 describes $ParA_\tau$ in detail.

Polyhedral optimization is a program optimization typically focusing on *loop nests*. Intuitively, a loop nest is represented as a multi-dimensional polytope representing its iteration space, as well as a separate polytope describing dependences between individual iterations. *Linear optimization* is used to determine a scheduling function mapping the original to a transformed iteration space while respecting all dependences. Typical optimization goals of linear optimization include minimal execution time, improved data locality, and minimal communication volume. [details on polyhedral optimization can be found in the seminal work of Feautrier [25, 26] and Lengauer [27]].

Post-order numbering refers to assigning each node of a directed graph a unique integer number (*post-order number* or *post-order ID*) while traversing the graph in a depth-first-search. The ID of each node is computed, once all successors of the node have been traversed, as the *last assigned ID* $+ 1$. In the context of this thesis traversal starts at the unique entry node (*CFG*) or *root* node (*PDG*); IDs start at 0.

Post-order traversal refers to traversing the nodes of a directed graph (in the context of this thesis a *control flow graph* or *program dependence graph*) in order of their *post-order numbers*.

Privatization in the context of this thesis refers to assigning disjoint copies of a variable or data structure to different and possibly concurrently executing pieces of code to

avoid interference. Privatization is an important parallelization-enabling technique used to resolve data dependences under certain conditions. Subsection 5.2.2 provides the definition of privatization used in this thesis.

Profiling refers to a *dynamic* analysis collecting information on the running application. In this thesis, profiling is used to collect call-site execution times (see Subsection 7.1.1 for details) and branch profiles (see Subsection 7.1.2 for details). The latter are used to estimate the frequency in which certain branches in the *control flow graph* are taken and how many iterations are executed on average per invocation of a given loop (loop trip count).

Program dependence graph (PDG) is the union of two sub-graphs sharing the same nodes: the *data dependence* subgraph and the *control dependence* subgraph. Like in the *CFG*, *basic blocks* or instructions form the nodes of the graph. Unlike the CFG, the PDG represents real dependences and is free from artificial control flow making it particularly suitable to reason about parallelism. Subsection 5.1.1 describes the form of PDG used in this thesis in detail, while Ferrante et al. [28] provide the classical definition and background.

Reduction, like *privatization*, forms an important *parallelization* enabling technique. The term *reduction* refers to a computational pattern that reduces the dimensionality of an input using a commutative and associative operation, for instance summing up all elements of an array of integers. Once such a pattern is recognized, a parallelizing compiler can make use of the associativity and commutativity properties to transform the induced dependences and introduce parallelism. Section 3.1 provides an overview of research work on reduction while Subsection 5.2.1 formally defines the notion of a reduction as used in this thesis.

Recursion in the context of programming languages and compilers describes a function being defined in terms of itself. In a program this manifests in a function F directly (by itself containing a call to F again) or indirectly (by calling another function which in turn transitively calls F) calling itself. Recursion is used to break down a problem into smaller sub-problems (for instance sorting arrays being defined based on sorting sub-arrays) and can be used to emulate loops. Consequently, recursive

algorithms/functions are a target of parallelization which is just as interesting as loops.

Runtime systems are used to observe and control the execution of an application. Typical tasks of a runtime system include resource management, garbage collection, scheduling, interpretation, and just-in-time compilation. The main tasks of $ParA_{\gamma}$'s adaptive runtime system are described in detail in Chapter 7.

Sambamba is a compiler framework for static/dynamic optimization based on the *LLVM* compiler infrastructure and one of the contributions of this thesis. Chapter 4 and Section 8.1 describe *Sambamba* in detail.

Sampling is a technique used to reduce the overhead of *profiling*. Instead of incurring the overhead of profiling in every instance of the profiled behavior (e.g., call-site execution times), only every *n-th* instance is profiled or only once every n milliseconds. Sampling can be used to trade accuracy for reduced overhead.

Sequential execution refers to the non-concurrent in order execution of a list of tasks.

Speculation is a parallelization enabling technique which allows to speculatively ignore certain conservatively assumed dependences, provided measures are taken to guarantee correct execution in case of a misspeculation. Speculation is used as a possibility to enable parallelization in this thesis (see Section 4.2 and Subsection 5.2.3). The implementation of the speculation mechanism is taken from Hammacher [29] where it is described in thorough detail.

Static single assignment (SSA) form is a property of a compiler *IR* with the purpose to make the *def-use* relation explicit and to simplify many analyses and transformations performed during compilation. Properties dictated by the definition of SSA include a single static definition of each value and each usage of a value being *dominated* by its definition. [see Cytron et al. [21] for details on SSA and ϕ-nodes].

Static in the context of this thesis refers to any information about an application won without executing it, i.e., by only inspecting a representation of the program itself. This is in contrast to *dynamic* information gathered during the execution of an application.

Tail-call optimization refers to an optimization performed to save overhead in case of the last statement of a function F preceding the *return* is a call to another function G whose result will simply be returned by F. The overhead of performing the bookkeeping involved in an additional regular call can be saved by reusing the stack frame of the current execution of F and replacing the call by a simple *jump* (or unconditional *branch*). This optimization plays a special role in case the tail-call is a recursive call (a tail-recursion), when this optimization effectively transforms recursion into a loop.

Task parallelism generally describes the distribution of different tasks working on possibly disjoint sets of data to different compute resources for parallel execution. This is in contrast to data parallelism in which the same task is executed in parallel on disjoint input data. Task parallelism in principle does not pose any restrictions on the form of the tasks and in particular does not depend on the concept of a loop.

Termination effect refers to the effect of terminating the currently executed function (Note that this definition also covers the termination of the whole application).

Chapter 3

State of Parallelization Research

This chapter gives a short overview of the very broad field of parallelization and, because of its importance to parallelization, reduction research. Due to the size of the field, this study of related work cannot claim to be complete. Instead it introduces several approaches to parallelization and reduction which have influenced modern automatic parallelization and, in many cases, are still cited today. It hints at problems and restrictions which have until today hindered broad adoption of automatic parallelization and shows open issues which motivate the work conducted as part of this thesis.

3.1 Reduction

In [30] Midkiff has summarized fundamental compiler techniques used in the context of automatic parallelization. One very important parallelization enabling technique is the exploitation of reductions which Midkiff defines according to the frequently used informal and syntactical formulation: *"[...] a compiler essentially looks for statements of the form 's = s ⊕ expr'. [...] the value of expr must be the same regardless of the loop order it is evaluated in. [...] the left-hand side s must not be used in other statements."* This definition, which in one form or another appears frequently in the literature, has two

important restrictions: It is tied to reductions being part of a loop; and it is solely based on a tight syntactical pattern. Syntactically different but semantically equivalent forms are not covered by this definition. Furthermore, a syntactical approach needs to run very early in the compiler toolchain. We will argue in this thesis that parallelization, and with it also reduction recognition and realization, need to be fully integrated into the compiler toolchain.

Dynamic approaches like *Privateer* by Johnson et al. [31] or the frequently cited *LRPD-test* by Rauchwerger and Padua [32] avoid reliance on statically analyzable reductions of a particular syntactic form. These approaches optimistically detect candidates of reduction operations and parallelize their containing loops. To guarantee correctness, the optimistical assumptions need to be dynamically validated: Rauchwerger and Padua propose to use shadow memory to keep track of dynamic accesses performed during the execution of the optimistically DOALL-parallelized loop. The reduction is validated a posteriori and execution of the loop completely repeated sequentially (i.e., non-speculatively) upon violation of the assumptions. The *extended reduction statements* of Rauchwerger and Padua [32] share properties with our data-flow based approach to reduction (see Subsection 5.2.1) but have no notion of the overhead introduced by a realized reduction, in particular of varying reduction locations.

In their follow-up work on *R-LRPD* (recursive LRPD) Dang et al. [33] refine the scheme and allow to re-execute the iterations succeeding the misspeculation again speculatively upon recovery instead of having to sequentially reexecute the whole loop. This changed approach is in favor of loops showing low but non-zero misspeculation rates which could not be profitably parallelized using the original approach. The overhead induced by the speculative reduction is discussed but not explicitly modeled by the authors. Making a qualified choice between different reduction and parallelization opportunities as for instance done by our approach in the *BiCG* example (Figure 1.4a) is not addressed.

Apart from the a posteriori validation of speculative assumptions on reductions Rauchwerger and Padua [32], as well as Dang et al. [33], describe an alternative validation method based on the inspector/executor principle: In [34] Rauchwerger and Padua describe an inspector loop which is generated by the compiler preceding the speculatively executed

loop to dynamically validate speculative assumptions on the memory access patterns. Depending on the outcome of the validation the candidate loop is executed sequentially, parallelized, or according to a schedule generated by the inspector (see [35] for this extended approach). The drawback of the inspector approach is obvious: a separate loop with the same iteration range is generated performing the same memory accesses as the original loop to determine the legality of the assumptions made. This not only requires all memory accesses to be loop-invariant (i.e., not altered by executing them twice), but is also costly. The authors propose to decide on a case-by-case basis if a separate inspector loop should be generated or if the checks should be performed after speculative execution of the loop. A semi-automatic selection scheme in the context of speculative reductions has been proposed a decade later by Yu and Rauchwerger [36]. The selection in this scheme is based on hardware dependent experiments performed on training data collected in synthetically generated benchmarks which are specific to a given problem domain.

A restriction which is shared by all approaches in the *LRPD* domain described above is their specialization to loop-based reductions performed on arrays. The validation of assumptions is based on shadow arrays being addressed by the indices of the array accesses performed during loop execution.

Recent approaches to reduction like the one of Ginsbach and O'Boyle [37] use constraint solving to improve the reduction detection capabilities abstracting completely from the syntactic form of a reduction and instead basing the recognition on semantic properties. Still the approach relies on unnecessarily narrow code features like considering only *for*-loops with a loop-invariant iteration range, for instance. Furthermore, it does not take into account the profitability of a possible exploitation of the identified reduction.

The approach of reduction recognition described in this thesis (see Subsection 5.2.1) is not as restrictive as the usual definitions. Nevertheless, it is by definition a static approach and consequently not able to detect all possible reductions.

In this comparison of reduction approaches we left out the excessive body of work on reduction (or recurrence) detection and realization in the context of scientific codes, as most approaches require the target code to work on regular data structures with access functions to be expressible as linear or affine functions. Our approach explicitly targets general

purpose codes with irregular and statically unpredictable, possibly also loop-invariant memory access patterns. For a summary on approaches to reduction detection, modeling and optimization in the scientific domain, please refer to Doerfert et al. [16].

3.2 Manual Parallelization

Using manual parallelization domain experts and proficient programmers can achieve the highest performance benefits in most cases. This in particular holds true for general purpose or irregular applications (in contrast to scientific or numerical applications), whose irregular control flow and unpredictable memory access patterns severely hinder automatic parallelization and necessitate domain knowledge to successfully exploit the inherent parallelism of an application. Libraries like pThreads [38], Java Threads [39], Intel TBB [40], language extensions like OpenMP [6, 7] or Cilk [41] and Cilk+ [42] and language built-in functionality like actors in Erlang [43, 44], Go [45] and Scala [46, 47] are used for manual parallelization at different levels of abstraction. Domain experts can make use of highly specialized domain specific languages like the Halide DSL for image processing of Ragan-Kelley et al. [48], that ships with a parallelizing compiler which is able to exploit high level features of the DSL to implicitly and explicitly encode parallel execution schedules. Unfortunately, defining such schedules generally requires a deep understanding of the performance implications of the algorithm at hand, or the capabilities of the execution platform. It requires a level of understanding which is beyond the capabilities of most regular developers: manual parallelization is hard and error-prone and most developers still say about themselves that they are unable or unwilling to deal with manual parallelization [4, 5].

One possibility of reducing the complexity of parallelization is to at least automate the identification of code regions amenable for profitable parallelization while still leaving the realization of the found parallelism to the developer. This approach was chosen by Mak and Mycroft [49] (c.f., [50, 51]) in their Embla tool, whose idea it is to observe the dynamic dependences between instances of method calls and loop iterations. By then employing a typical critical path analysis the authors are able to pinpoint and propose candidate regions for parallel execution. As the proposal is based on dependences that manifested

during chosen profiling runs, it may be unsound. Validation, and actual parallelization, consequently has to be done manually or a potentially costly speculation system has to be used to guarantee correctness. The principle idea of Embla and Embla2 is similar to our very own previous work [20]. The ParaMeter tool of Kulkarni et al. [52] similarly identifies the data parallelism of irregular applications.

The goal of parallelization as described in this thesis is to be generally applicable. It will in most cases not achieve the performance gains achieved through manual parallelization done by a domain expert, but it enables a broader range of developers to parallelize their applications. Even if automatic parallelization is not possible, our approach to semi-automatic parallelization as described in Chapter 10 provides a helping hand and a safety net to non-expert programmers who will in turn provide domain-specific knowledge, which is hard or impossible to automatically deduce, to our parallelizing compiler.

3.3 Static Parallelization

Burke et al. [53] describe the exploitation of nested fork-join parallelism while taking into account the possibility to resolve (or eliminate) data dependences by using privatization. Parallelism is greedily introduced in the form of *DOALL*-loops and *COBEGIN... COEND* blocks of parallel processes. The approach does not trade parallelism for overhead, parallelizes everything it can, and uses all opportunities of privatization to increase parallelism without taking profitability or available computational resources into account. The model of parallelism in that work, which is described as being "general and simple", shares important properties with the model described in this thesis, but is less expressive. We furthermore do not greedily introduce parallelism but instead take the introduced overhead into account when statically and dynamically striving for profitable parallel execution.

In a similar fashion, Sarkar [54] presents a heuristics-based approach to statically parallelize task trees computed from the program dependence graph of FORTRAN functions. The approach takes into account the overhead introduced by parallelization as well as profiling information collected during dedicated executions of the target application to statically

estimate the profitability of the parallelized code. The enforced tree structure, motivated by the requirement to generate a parallel FORTRAN program with structured parallelism, limits the flexibility of the approach. The linear-programming-based scheduling of hierarchical task graphs for embedded systems by Cordes et al. [55] shares this limitation and further imposes restrictions on the shape of the generated parallel code regions.

Rugina and Rinard [56] propose a simple method to automatically parallelize divide-and-conquer algorithms as a use-case to their sophisticated region-based inter-procedural memory analysis: parallelism is introduced to a *C* program in the form of a *Cilk spawn* instruction preceding every relevant call-site, and a *Cilk sync* succeeding it. The parallel region formed this way is then expanded by moving the *sync* along the control-flow path until the dependence analysis forbids further propagation because of a potential conflict of the next statement with the spawned function. While the dependence analysis proposed in the work by Rugina and Rinard is quite strong, in particular for regular data structures, the simple parallelization approach is very limited as it is unable to abstract from the implemented control-flow.

Zhong et al. [57] describe an approach of automatic speculative DOALL parallelization of loops relying on hardware transactional memory, hardware-based low-cost thread spawning and low-latency inter-core communication. Mehrara et al. [58] implement a software transactional memory system to get rid of these hardware requirements. The described STM is specialized and limited to automatic DOALL parallelization of loops, however. Kim et al. [59] apply speculative DOALL parallelization to distribute the computation performed by a loop to a cluster of machines.

Madriles et al. [60] propose Anaphase, a fine-grained speculative parallelization technique finding regions for parallel execution and scheduling the code using a multi-level graph partitioning approach. The approach speculatively parallelizes a given sequential application at the level of single instructions driven by several heuristics estimating the affinity of computation nodes. Anaphase relies on hardware support for efficient recovery from misspeculation.

Suesskraut et al. [61] introduce Prospect, a compiler framework using an approach which they call *predictor/executor*, which resembles the *master/slave* parallelization concept of

Zilles and Sohi [62] by providing a fast but potentially incorrect variant and a slow but correct variant of the code. Parallelism is introduced by executing the fast variants on the critical path, and multiple slow variants in parallel to verify the results of the fast variants. The approach introduces very high overhead: even in the best case, i.e., in case no roll-backs occur, the slow variants have been occupying more computational resources than the actual (fast) computation.

To optimize loop nests, in particular for mathematical and scientific applications mostly based on the usage of regular data structures and control flow, so-called polyhedral loop nest optimization has been proposed by Feautrier [25] in his seminal work on scheduling in the polyhedral model for one-dimensional [25] and multi-dimensional [26] time. The focus of that work has been on efficient scheduling, while parallelization has been described as one possible use-case. Later work by Lengauer [27] and Feautrier [63] specifically dealt with parallelization based on polyhedral scheduling. Pluto by Bondhugula et al. [64] is a *C* source-to-source compiler which uses polyhedral scheduling to produce a parallelized *OpenMP* [6, 7] program. Pluto is able to achieve extreme performance by far outperforming state-of-the-art productive compilers, if, and only if, the polyhedral model is applicable at all, which still is a drawback of polyhedral optimization. The cost of using this very clean and elegant mathematical model is a limited applicability with respect to irregular applications. A loop, or more precisely a static control part (SCoP), represented in the model typically needs to fulfill certain criteria: loop bounds as well as the predicates of conditionals used in the loop body have to be representable by affine functions in the surrounding loop indices as well as (provably) loop invariant parameters. Dependences between individual statements are only allowed via accesses to indexed variables (arrays), whose access functions are affine, also in the above mentioned parameters. Furthermore, called functions need to be statically known and provably pure[1]. These are severe restrictions whose mitigation has been the goal of excessive research work [65–70], conducted also by ourselves [16, 18] and Doerfert et al. [71]. Parallelization in the polyhedral domain is related but not addressed by the work described in this thesis. Its mathematically clean representation and optimization-based scheduling however have had a strong influence on our work.

[1] A pure function does not have any observable side-effect, and it computes the same result when called with the same arguments, i.e., it is independent of any hidden state.

Decoupled Software Pipelining (DSWP) aims at parallelizing sequential loops by forming patterns of pipelined execution [72]. Loops are decomposed into pipeline stages, possibly executing in parallel to each other. Each stage communicates produced values to the threads executing later stages as needed. DSWP has been extended in multiple ways over the years: Ottoni et al. describe how to automatically perform thread extraction [73]; the work of Raman et al. allows to distribute a single pipeline stage to multiple threads [74], introducing further parallelism. The work of Vachharajani et al. describes how to speculatively parallelize [75], and August et al. enables cross-invocation parallelism among iterations of different loop instances [76] for loops of a specific shape. Huang et al. [77] generalize the idea of Raman et al. [74] and enable the parallelization of individual DSWP stages by manually applying a secondary loop parallelization scheme. The work clearly shows that different parallelization schemes can be profitably combined. However, the question on how to automatically select and prioritize different approaches is considered to be a challenging open research question by the authors. While modern implementations of DSWP, like *Parcae* [78] for instance, avoid it, the earlier approaches rely on specialized hardware for inter-thread communication and recovery from misspeculation. The approach described in this thesis instead runs on commodity systems.

Vandierendonck et al. [79] (also [80]) describe *Paralax*, a semi-automatic approach of parallelization in a DSWP like fashion. Like the approach presented in this thesis the approach relies on DSA [23] for its dependence analysis, and suffers from the same imprecision as we do. To address this concern, Vandierendonck et al. [79] motivate a set of user annotations, which gave partial inspiration to our approach of semi-automatic parallelization presented in Chapter 10.

In Helix [81], adjacent loop iterations are automatically distributed in a round-robin fashion to different threads executing on adjacent cores of the same processor. The latency of inter-core communication necessary to transfer values to fulfill loop-carried dependences is hidden by exploiting the SMT capabilities of modern multi-core processors: potentially needed values are continuously pre-fetched to guarantee their availability in the local L1-cache without latency once they are used by the target core. While the performance results are impressive, the authors show in their own follow-up work [82] that the approach does not scale to more than four cores and propose hardware support in the form of

a proactive ring-cache interconnecting all participating compute cores to overcome this limitation by being able to send values from one core to the next with a delay of one clock cycle. Both approaches are limited to parallel execution of a single loop on the cores of a single processor at a time.

3.4 Runtime-centric Parallelization

Kulkarni et al. [83] require the programmer to use the graph data structures and iterators provided by their Galois system in order to make dependences between graph nodes explicit. In return, these explicitly stated dependences enable dynamic parallel execution of graph-based algorithms without the need for conservative assumptions.

Not limited to graph-based algorithms *PetaBricks* as proposed by Ansel et al. [84] similarly allows to explicitly express the dependences in the code using a specially designed implicitly parallel programming language. Like Galois *PetaBricks* allows to make dependences explicit and automatically selects among and switches between multiple user-defined alternative algorithms and tunes user-defined parameters. The experiments presented in [84] show how platform-specific and input-dependent the performance gains achieved by parallelization are and support our dynamic approach. While the auto-tuning capabilities of *PetaBricks* exceed those of the approach developed in this thesis, it requires the use of a special language and to have non-negligible domain expertise to be effective.

Out of order Java by Jenista et al. [85] and later improved by Eom et al. [86] provide a task extension to the Java language allowing the programmer to mark regions of the code to be considered for parallel execution. The compiler generates lightweight runtime checks, enabling efficient pre-validation of potential conflicts at runtime before spawning a parallel task. Both approaches rely on the programmer to rethink and rewrite the subject application.

Chen and Olukotun [87] implemented a runtime system for Java applications that dynamically monitors dependences between loop iterations. To find promising parallelizable loops, the approach relies on a hardware profiler. DeVuyst et al. [88] and Hertzberg and Olukotun [89] followed a similar idea. By employing runtime binary translation, their

approaches do not rely on the availability of the application source code. The approach relies on runtime performance monitoring implemented in hardware to efficiently refrain from parallel execution in case of non-profitability. Johnson et al. [90] makes heavy use of thread level speculation supporting hardware to empirically optimize an application after a profiling run preceding the actual execution. All these approaches rely on special hardware in contrast to the work presented in this thesis.

To soften the requirement of typical polyhedral optimizations to statically prove functions affine, Jimborean et al. [69] (also [70]) statically speculate on the linearity of loop bounds and memory accesses and generate corresponding code skeletons using the classical polyhedral techniques assuming linearity. At runtime the accesses are monitored and linearity validated. Execution can proceed speculatively and is rolled back upon violation of the statically made assumptions. The Approach of Jimborean et al. [69] shares many ideas with ours, in particular statically performing costly analyses to identify and prepare optimization candidates whose instantiation is left to the runtime system. Pradelle et al. [91] extend *VMAD*, the approach of Jimborean et al. [69], by parallelizing the binary at runtime. Baghdadi et al. [68] seek to extend the applicability of the polyhedral model by dynamically verifying statically made assumptions, which enable to use the polyhedral model in the first place.

The *Parcae* system by Raman et al. [78] provides a flexible parallel execution environment and promises to allow for holistic optimization of a parallel program instead of mere parameter tuning, as for example done by Karcher and Pankratius [92] in the context of parallelization. Parcae relies on extensions to the operating system to orchestrate the parallel execution of different applications.

To further stretch the limits of automatic parallelization without resorting to expensive speculation mechanisms the authors of *Helix* [81] make use of a relaxed program semantics in their *Helix-UP*-approach [93] ("UP" is for unleashed parallelism). The latest incarnation of Helix [94] uses speculation in the form of a software transactional memory system. Like *Sambamba*'s own STM-based approach to speculation of Hammacher [29], the approach of Murphy et al. [94] uses TinySTM and confirms that it is crucial to guard only small code

sections (sequential segments in their parlance) to allow profitable parallelization using STM.

3.5 Conclusion and Open Issues

A tremendous body of research work in parallelization has been created in the last decades, which is still being extended. The above mentioned approaches can only be understood as a small excerpt taken from the field. Two families of approaches, however, seem to stand out, in particular when it comes to parallelizing general purpose applications: the decoupled software pipelining (DSWP) family of approaches who seek to exploit a pipelined parallel execution model of loops, which limits the available parallelism to the number of available pipeline stages. And the Helix family of approaches which seeks to push the boundaries of DOACROSS style loop parallelism in the presence of loop-carried dependences, which are sought to be satisfied by extremely lightweight inter-core communication and synchronization. The scalability limiting factor of Helix is the communication latency. Both families are, like many others, by design limited to parallelizing loops only.

Also in line with other researchers, later approaches in both families rely on speculation or other forms of mitigating the necessity to rely on statically made conservative assumptions on the existence and manifestation of control and data dependences. Independent of DSWP or Helix, it is in the context of general purpose applications with irregular data-structures and dependences commonly agreed upon that conservative static analysis alone cannot be the driving force of automatic parallelization for modern multicore and manycore systems. The consequently necessary runtime parallelization, speculation, and similar measures today require a complex runtime system to orchestrate the parallelism whose profitability still is nearly as unpredictable statically as is its existence. The oftentimes assumed hardware support for efficient profiling, speculation and near zero-cost inter-core communication is not generally available yet, though speculation in particular finds its way into commodity hardware.

While several approaches of automatic parallelization exist, which work particularly well for the program patterns they have been specifically designed for, a joint cost-model is lacking that is able to drive the selection between different approaches applicable to a given loop or code region. Even worse, most approaches assume the sole control over the parallel execution of the whole application which raises the question of compatibility between different approaches, in particular facing the above mentioned complex runtime systems. Consequently, a joined cost model would not only be required to select one approach for a given loop, but one approach and its runtime system for the whole application. Given the oftentimes narrowly defined target code patterns of the most promising parallelization approaches, a combination, or, even better, a generalization of different parallelization approaches is a worthy research target that we address in this thesis.

Chapter 4

Sambamba — A Static/Dynamic Parallelization Framework

This chapter introduces the general parallelization and optimization framework *Sambamba* on a conceptual level. Its conceptualization and implementation are an essential part of this thesis work and its contribution. Building on top of *Sambamba*, $ParA_\tau$, a simple task-based parallelization approach is introduced, which forms the conceptual and technical basis of $ParA_\gamma$, our final approach to generalized task parallelism. To not distract the reader, technically interesting but conceptually less important details are left for Chapter 8.[1]

Sambamba provides a reusable and extensible framework for online adaptive program optimization with a special focus on parallelization. To avoid being dependent on a particular programming language or even processor architecture, *Sambamba* is based on the LLVM compiler infrastructure [95] and consists of a static part (compiler) and a runtime system. The framework is organized in a modular way and can be extended by adding so-called modules. Such modules consist of two parts: *compile-time parts* handle costly analyses such as inter-procedural points-to and shape analysis as used by our parallelization module. These results are fed into the *runtime parts*—analyses conducted at runtime which are able to adapt the program to runtime conditions and program inputs. Obviously,

[1] Design and implementation of the *Sambamba* framework have been done together with my colleague Clemens Hammacher. $ParA_\tau$, which is the subject of Section 4.1, has been completely conceptualized and implemented by myself as part of this thesis work.

it is crucial for the runtime analyses to be as lightweight as possible. The main task of the *Sambamba* runtime environment is to manage and separate different registered modules and provide facilities for selective re-compilation of parts of an application. Furthermore, the framework provides facilities to carry over analysis results from the compile-time parts of a module to the runtime parts.

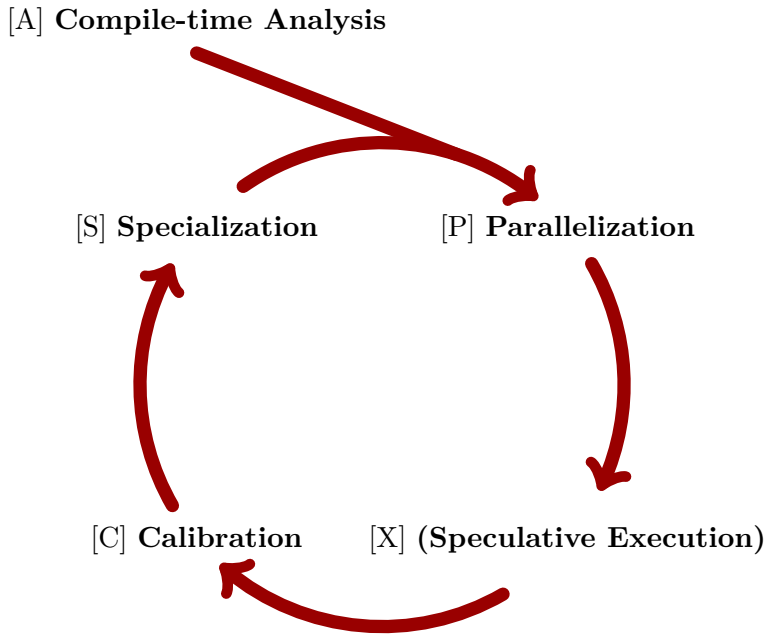


FIGURE 4.1: *Sambamba* execution steps.

The high-level flow of execution in the *Sambamba* framework is depicted in Figure 4.1:

[A] We use static whole-program **analyses** to examine the program for potential optimizations and propose a first set of parallelization and specialization candidates that are deemed beneficial. For long-running programs it might be a viable alternative to also run these analyses at runtime.

[P] The runtime system provides means for speculatively **parallelizing** parts of the program based on the initial static analysis and calibration information.

[X] We detect conflicts caused by speculative **executions** violating the program's sequential semantics and recover using a speculation system. Two different speculation systems are implemented in the *Sambamba* framework.

[C] We gather information about the execution profile to **calibrate** future automatic optimization steps. Speculative optimization can also gather information on misspeculation rates in this step to guide further decisions.

[S] *Sambamba* supports generating different function variants based on the results of the calibration phase. Such variants are **specialized** for specific environmental parameters and input profiles. These can then again be individually parallelized in the next round of adaption.

Section 4.1 and Chapters 5 to 7 explain two parallelization approaches implemented based on the *Sambamba* framework in detail.

4.1 Simple Task-based Parallelization — $ParA_\tau$

In this section, we describe $ParA_\tau$, a first task-based parallelization scheme implemented on top of the *Sambamba* framework. This scheme is conceptually simple, but it shares the technical basis whose further development lead to $ParA_\gamma$, the approach described in detail in Sections 5 to 7. Where appropriate, this section is used to introduce these foundational techniques.

We call $ParA_\tau$ task-based because it seeks to parallelize generic code regions, independent from loops. It has been motivated by the work of Rugina and Rinard [56]. The approach is unable to parallelize loops, but is also not limited to, although certainly well suited for, parallelizing divide and conquer algorithms. In contrast to many actively developed and researched parallelization schemes, we neglected loop parallelization at first, and concentrated on task parallelism instead, which seems to be mostly ignored by modern research in automatic parallelization.

As a simple example consider the code in Figure 4.2 (The full sources can be found in Appendix A). The *performTask* function is doing the main work and will be parallelized by the approach. *performTask* first recursively constructs two linked lists (X and Y), performs some heavy computation (*hashList*) requiring to traverse the list and touch each element, and finally recursively frees the elements of both lists before returning the

```
1   typedef struct list {
2     struct list *Next;
3     int Data;
4   } list;
5
6   /*
7    * Definitions for methods makeList, hashList and freeList
8    * omitted. Please refer to Appendix A for the full sources.
9    */
10
11  long performTask(int size) {
12    list *X = makeList(size);
13    list *Y = makeList(size);
14
15    long hash_X = hashList(X);
16    long hash_Y = hashList(Y);
17
18    freeList(X);
19    freeList(Y);
20
21    return hash_X * hash_Y;
22  }
23
24  struct timeval start, end;
25
26  int main() {
27    while (1) {
28      gettimeofday(&start, 0);
29      long res = performTask(1 << 10);
30      gettimeofday(&end, 0);
31
32      double secs = (end.tv_sec - start.tv_sec) +
33                    1e-6 * (end.tv_usec - start.tv_usec);
34
35      printf("result after %5.2f seconds: %ld\n", secs, res);
36    }
37
38    return 0;
39  }
40
```

FIGURE 4.2: Simple application containing irregular data structures and recursive functions.

result. While this is only a toy example, the difficulty for most automatic parallelization approaches lies in the fact that the application heavily relies on so-called irregular data-structures (*list*) and recursive functions (*makeList*, *hashList*, *freeList*), which are both techniques potentially used in general-purpose applications.

4.1.1 Dependence Analysis

An integral part of each parallelization scheme is the underlying dependence analysis, as dependences are the one limiting factor of parallel execution: if one computation B

depends on the results of a computation A, then (without relying on speculating what the result will be) it has to wait for the result to be available before starting execution.

Another form of dependence is introduced by externally observable behavior and the requirement not to violate the sequential semantics of a program: except for the timing behavior, an application should have the same observable effects before and after parallelization. Parallelization is a very aggressive optimization, but usually it is required not to change the semantics of an application.

In this work, we base our dependence analysis on the *DS-Analysis* (or *DSA*) by Lattner et al. [23]. *DSA* has been designed as a scalable points-to-analysis in the *LLVM*-Framework. It has several important properties of which we mention the ones which are of particular importance in the context of this thesis. Furthermore, to improve the precision for our use-case, we had to rethink a few compromises which the authors of *DSA* made to improve scalability. Important properties, as well as necessary changes we made to the *DSA* are explained in the following paragraphs.

The first very important property of *DSA* is its **flow-insensitivity**. The result of the analysis is a so-called *DS-Graph* ($DSG[f]$) per function f describing the effects of this very function to local memory objects, which form the nodes (*DS-Nodes*) of the graph. This is in contrast to flow-sensitive analyses which deliver for every individual program-point/instruction the respective effects. This is a trade-off between precision and speed that we accepted in the name of reasonable scalability.

Second, *DSA* is **context-sensitive**, which means that it is able to take calling contexts into account when analyzing a function. Technically this is implemented by *DSA* in the last of three phases: *DSA* is separated into a local phase, a bottom-up phase, and a top-down phase. Each phase results in a *DS-Graph* per function, which contains a node for each static virtual memory cell (a piece of memory allocated at once, *static* because *DSA* does not distinguish between dynamic instances of an allocation). Furthermore, it contains a node per virtual register (one could say “variable”) used in the code, linked by an edge to the memory block the register might refer to during the course of execution. Memory cells in the *DS-Graph* are connected to each other to resemble the statically analyzable pointer structure among the cells.

The **local phase** of *DSA* computes the local effects of a function without taking called functions or any calling context into account. It does so by accumulating the effects of individual instructions. Details are described in the original paper, but one relevant property in this step worth mentioning is **unification**: When *DSA* encounters a program point at which a virtual register (or variable) used in the code might point to a different memory location than assumed so far, the two memory cells in question are unified in the *DS-Graph*. From that point on, those cells and their respective properties are indistinguishable from *DSA*s perspective. Consider the simple *min* function to the right which takes two pointers to integers and returns the one that points to the smaller one. Due to the unification, *DSA* is unable to distinguish between the cells pointed to by the variables *res*, *a*, and *b*. After processing line 5, *DSA* would unify the cells pointed to by *res* and *a*, after processing line 7, it would also join the cell pointed to by parameter *b*. This is an important restriction. In particular, this unification of parameters propagates in the following bottom-up phase to the callers' respective graphs. We refer to the *DS-Graph* of a function f resulting from the computations of the local phase as $DSG_\alpha[f]$.

```
1  int *min(int *a, int *b) {
2    int *res;
3
4    if (*a < *b)
5      res = a;
6    else
7      res = b;
8
9    return res;
10 }
```

In the second, the **bottom-up phase**, *DSA* merges *DS-Graphs* of callees into the respective *DS-Graphs* of all calling functions. It does so by mapping function parameters and return values of the callee graph to the corresponding *DS-Nodes* representing the pointer-compatible arguments and return value of the function call instruction in the caller graph and inlining other reachable nodes into the caller graph. This step also leads to unification of *DS-Nodes* and consequently potential imprecision of the analysis. Again, this is a trade-off that we accepted. The bottom-up graph of a function f is referred to as $DSG_\perp[f]$.

The last phase is the **top-down phase**, which works similarly to the bottom-up phase and is the source of context sensitivity. Results of the caller graphs regarding the arguments (the *context*) of call instructions are pushed into callee graphs. The top-down graph of a function f is referred to as $DSG_\top[f]$.

Context-sensitivity is dropped by *DSA* for strongly connected components (SCCs) of the call graph, i.e., recursion-induced cycles in the call relation between functions.

One trade-off which we did not accept for our parallelization is the fact that *DSA* calculates a single so-called **globals graph** which contains *DS-Nodes* for all global variables of the whole program. In contrast, *DS-Graphs* of individual functions do not contain global variables at all. This decision has been made to not having to replicate effects on globals into each and every *DS-Graph* computed. In our context however, this behavior in combination with unification results in an unacceptable imprecision concerning code working on global variables. Imagine, somewhere in the whole application, the minimum over two globals is computed by the function shown above. Those two globals would be indistinguishable and any two pieces of code working on one of them would be the sources of a mutual dependence.

Instead of relying on a single globals graph, we adapted the *DSA* to treat globals just like any other value. In the bottom-up phase however, *DS-Nodes* representing globals are inlined from the callee's *DS-Graph*s into the caller's graphs, risking excessive propagation of globals among all graphs. This however is necessary to make a *DS-graph* of a function as it results from the bottom-up phase represent all possible effects a called function might (transitively) have, unifying global nodes only where it is dictated by the transitively called functions.

Figure 4.3 shows $DSG_\alpha[performTask]$, the *DS-graph* as computed by the local phase of the *DSA*. It contains an elliptical node per virtual register (*%X* and *%Y* corresponding to the variables X and Y in Figure 4.2), each pointing to a virtual memory cell represented by a rectangular node with rounded corners, and a rectangular node per function call. Details on this representation can again be found in [23], but what you can easily see is that *DSA* is able to determine that X points to the value returned by *makeList*, which is later used as an operand of a call to *hashList* and *freeList* respectively. Furthermore, we see that from analyzing *performTask* locally, *DSA* concludes that the memory regions reachable from X and Y respectively are disjoint and not reachable from each other.

Figure 4.4 shows $DSG_\perp[performTask]$, the bottom-up graph of *performTask*. The calls have been processed by joining the information of their correspondingly called functions,

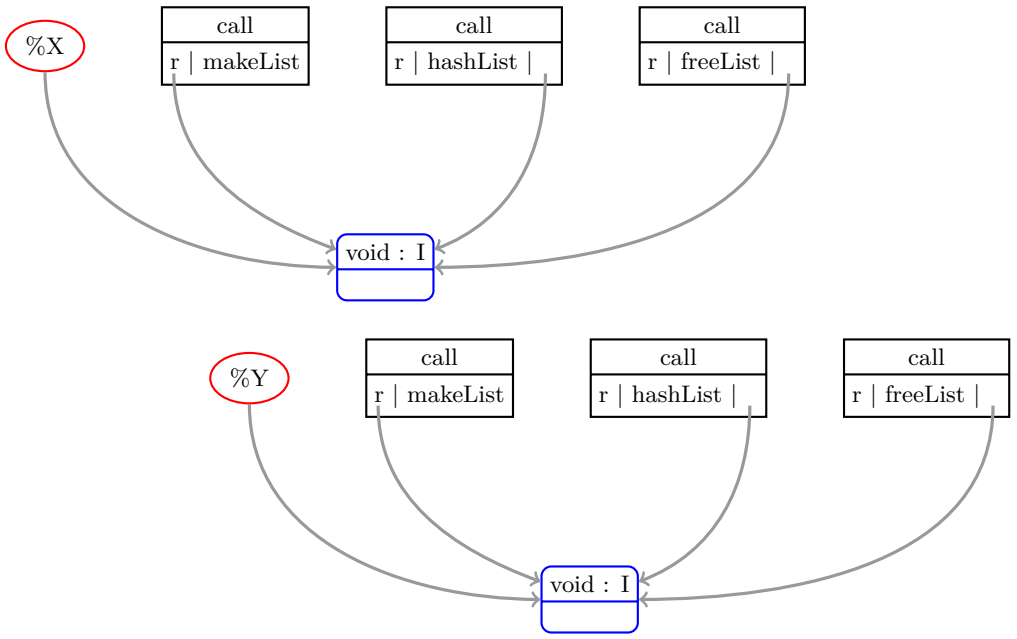


FIGURE 4.3: Local *DS-Graph* $DSG_{\alpha}[performTask]$ of the *performTask* method.

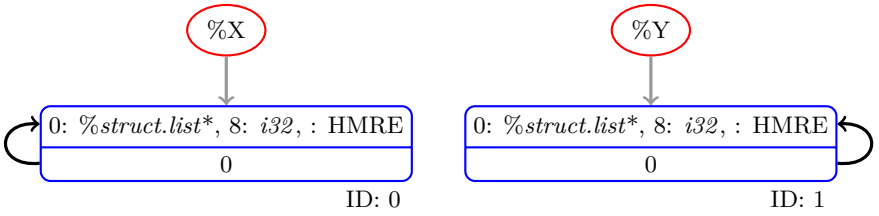


FIGURE 4.4: Bottom-up *DS-Graph* $DSG_{\perp}[performTask]$ of the *performTask* method. $DSG_{\top}[performTask]$ is equivalent as the context does not add information. Below right of each memory node you find the *DS-Node* ID as used by $ParA_{\tau}$ internally.

the call nodes are removed from the *DS-Graph*. *DSA* is now aware of the fact that values X and Y point to a list struct, which holds a pointer to another list struct, as well as a 32bit integer (*i32*) value. We still see that the lists pointed to by X and Y respectively are disjoint. This graph is equivalent to $DSG_{\top}[performTask]$ as the top-down phase does not yield any additional information.

When parallelizing a function f, $ParA_{\tau}$ now uses the information computed by the slightly modified version of *DSA* to compute the (abstract) memory effect of every individual instruction *insn* in the form of two bitsets $Eff_r[insn]$ and $Eff_w[insn]$ representing the read and written memory cells respectively. Memory cells are defined as the *DS-Nodes* of $DSG_{\top}[f]$, i.e., $Eff_r[insn]$ and $Eff_w[insn]$ contain as many bits as $DSG_{\top}[f]$ contains

DS-Nodes. Additionally, one bit is stored per instruction representing observable non-memory effects ($nme_r[insn]$ and $nme_w[insn]$ respectively). A non-memory effect is an externally observable effect, such as printing a log message to the terminal (non-memory write effect), reading the current system time (non-memory read effect), or asking for user input (non-memory write and read). Non-memory effects of the C standard library and known system-calls are hard-coded in our dependence analysis; statically unresolvable indirect and external calls are conservatively assumed to have non-memory read and write effects.

Finally, one bit $term[insn]$ represents the termination effect of an instruction. An instruction has a termination effect if it might terminate the execution of the containing function. This naturally holds true for a *return* instruction, but also for a call to *exit()* for instance, or a call to a function which might potentially (transitively) call *exit()*, effectively terminating the whole application. Note that $ParA_\tau$ has to be conservative: any indirect call which we cannot statically resolve or an external function call of which we do not have any further information has to be assumed potentially terminating.

$Eff_r[insn]$ and $Eff_w[insn]$ are computed depending on the semantics of *insn*, which in our case is an instruction of the *LLVM* intermediate representation. If the instruction is known not to touch any memory or have any side-effects (*ReadsNone* in *LLVM*), both effect sets are empty and the *nme* and *term* bits false respectively.

In all other cases, we take $DSG_\top[F]$ as the basis for the effect computation, where F is the function containing *insn*. We take the top-down graph as it represents the effects of the function to parallelize, taking into account all possible calling contexts. This way, the parallelized function will also be usable in all possible calling contexts. Note that in case the function F is externally visible, i.e., has external linkage, *DSA* has to be conservative and assume the worst with respect to calling contexts, as it simply does not see all possible callers. In that case, it has to assume, for instance, that all pointer and type compatible parameters might alias or be reachable from each other, which would result in unification of the corresponding *DS-Nodes* in $DSG_\top[F]$.

When inspecting an instruction *insn* $ParA_\tau$ (and also $ParA_\gamma$) computes the write (read) effect by taking the *DS-Node*s representing the written (read) *LLVM* values and all

DS-Nodes reachable from that one node and sets all bits corresponding to those nodes in $Eff_w[insn]$ ($Eff_r[insn]$), if, and only if the *DS-Node* is marked *written* (*read*) or *incomplete* in $DSG_\top[F]$. We also mark the reachable nodes, as we assume that an instruction that would change a pointer to a *struct*, for instance, conflicts with an instruction that writes to a member value of the referenced *struct* later on.

Call (and *invoke*) instructions are treated differently. If we cannot resolve the called function statically, we have to assume that the call might write and read all memory cells transitively reachable from the pointer compatible operands of the call, as well as all global variables. In that case, all corresponding bits are set in both effect sets. If, however we can resolve the call, we take $DSG_\bot[F']$, the bottom-up graph of the called function *F'* and compute the n-to-m[2] *callee caller mapping* of the pointer compatible operand and return values within $DSG_\top[F]$ and the corresponding nodes in $DSG_\bot[F']$. The effect sets of the call instruction are then marked just as described for the generic instructions, but instead of taking the read/write information from the reachable *DS-Nodes* in $DSG_\top[F]$, we take this information from their mapped nodes in $DSG_\bot[F']$. The rationale behind this approach is to be as precise as possible for call instructions which are a promising candidate for parallel execution. Taking the bottom-up graph of the called function allows to take only the specific calling context of the inspected *call* into account, which we do by computing the callee-caller mapping. Taking the top-down graph of the called function would be more conservative, as it takes all possible calling contexts into account, which is unnecessary, given the specific one we currently observe.

Figure 4.5 shows the effect bits per line of code of the *performTask* function. The IDs used to address the bits of $Eff_r[\cdot]$ and $Eff_w[\cdot]$ correspond to the bits annotated to the memory nodes in Figure 4.4. We can see that the two calls to *makeList*, as well as the respective pairs of calls to *hashList* and *freeList*, work on disjoint memory regions and that the return statement only has a function terminating effect $term[\cdot]$. Note that the *hash_X* and *hash_Y* values are kept in registers. Using those values does not have any memory effect.

[2] An n-to-m mapping might result from unification happening during the independent computation of the mapped graphs. This is also a detail that we had to add to *DSA* which assumed a 1-to-n mapping.

	$Eff_r[\cdot]$		$Eff_w[\cdot]$		$nme_r[\cdot]$	$nme_w[\cdot]$	$term[\cdot]$
	1	0	1	0			
long performTask(**int** size) {	-	-	-	-	-	-	-
list ∗X = makeList(size);	-	-	-	✓	-	-	-
list ∗Y = makeList(size);	-	-	✓	-	-	-	-
	-	-	-	-	-	-	-
long hash_X = hashList(X);	-	✓	-	-	-	-	-
long hash_Y = hashList(Y);	✓	-	-	-	-	-	-
	-	-	-	-	-	-	-
freeList (X);	-	✓	-	✓	-	-	-
freeList (Y);	✓	-	✓	-	-	-	-
	-	-	-	-	-	-	-
return hash_X ∗ hash_Y;	-	-	-	-	-	-	✓
}	-	-	-	-	-	-	-

FIGURE 4.5: The effect bits of *performTask* from Figure 4.2. *DS-Node* IDs correspond to those in $DSG_{\perp}[performTask]$ as depicted in Figure 4.4.

An effect set plus the non-memory and termination effects per instruction can typically be efficiently represented in the form of a single 64bit value. In the following and for the sake of a readable syntax we treat individual bits (e.g., $nme_w[\ldots]$, $nme_r[\ldots]$ and $term[\ldots]$) like truth values in logical formulas. Union and intersection of bitsets can be safely assumed to be implemented as bitwise *or* and *and* respectively.

A memory conflict $conf[i, j]$ of instructions i and j is now defined as follows:

$$conf[i,j] := Eff_r[i] \cap Eff_w[j] \neq \varnothing \tag{4.1}$$

$$\vee Eff_w[i] \cap Eff_r[j] \neq \varnothing \tag{4.2}$$

$$\vee Eff_w[i] \cap Eff_w[j] \neq \varnothing \tag{4.3}$$

$$\vee(term[i] \wedge (nme_w[j] \vee Eff_w[j] \neq \varnothing)) \tag{4.4}$$

$$\vee(term[j] \wedge (nme_w[i] \vee Eff_w[i] \neq \varnothing)) \tag{4.5}$$

$$\vee i \rightarrow_{du} j \vee j \rightarrow_{du} i \tag{4.6}$$

Equations 4.1, 4.2, and 4.3, represent accesses of the involved instructions to the same

(abstract) memory cell. Equations 4.4 and 4.5 check for potential function termination, which prohibits parallel execution of any externally observable effect and therefore causes a conflict. Finally, equation 4.6 triggers a conflict in case of a *def-use* relation between instructions i and j which means that one instruction uses a register value computed by the other instruction.

Based on the definition of a conflict, a dependence $(i \rightarrow j)$ between two instructions i and j is now defined as follows:

$$(i \rightarrow j) := \mathit{conf}[i, j] \wedge (j \ll^* i) \tag{4.7}$$

The $\ll$-relation is defined as a structural dependence or a follows relation. It is dictated by the program structure and corresponds to succession in the control-flow graph (*CFG*). $\ll^*$, the transitive closure of $\ll$, corresponds to reachability in the *CFG*. Intuitively this rules out dependences between two instructions that do not reach each other. Furthermore, it defines the direction of dependences: if an instruction j (transitively) succeeding an instruction i ($i \ll^* j$) is in conflict with i ($\mathit{conf}[i, j]$), we say "j depends on i" ($(j \rightarrow i)$) or j has a dependence on i. While this is basically a syntactical issue, we stress this as the literature does in no way agree on the direction of dependences. Figure 4.6 shows the *CFG* of the *performTask* method with structural dependences depicted as dashed (blue) arrows. Register or *def-use*-induced dependences are depicted as solid (gray) arrows. Memory-induced dependences are not shown.

For further processing, automatic parallelization for instance, the effect information is stored per instruction (or accumulated per basic block, depending on the granularity of parallelization). $ParA_\tau$ never stores actual dependences which are computed on the fly if and where necessary. This is in favor of being able to restructure/transform the code after the effects have been computed without the need to update dependence information. Inlining and code duplication in favor of parallel execution are two examples of transformations that profit from this design decision.

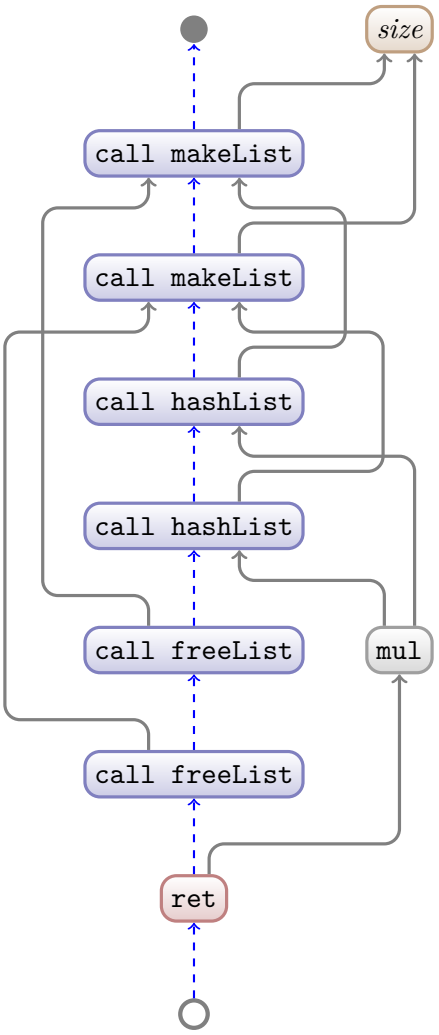


FIGURE 4.6: Regular CFG of the *performTask* method. Dashed arrows depict structural dependences, i.e., the order of instructions as dictated by the program structure, of instructions with possible side-effects.

We have chosen the *DSA* as the basis for $ParA_\tau$ because we primarily aimed at parallelizing general-purpose applications which, as described earlier, tend to make use of irregular data-structures for which the *DSA* was designed. However, we did so knowing that regular data-structures (i.e., arrays) are a weak spot of the *DSA* as it is not able to distinguish between individual array cells, except if the array size is statically known. Typically, an array is treated by *DSA* as a single virtual memory cell. While $ParA_\tau$ suffers from this restriction when dealing with code working on regular data structures, the dependence analysis of $ParA_\gamma$ has been extended to also deal with arrays, yet still irregular data-structures are also the main target of $ParA_\gamma$. $ParA_\gamma$ is explained in detail in Chapters 5

to 8.

4.1.2 Basic-block-wise Parallelization

Inspired by the work of Rugina and Rinard [56] and using the dependence information described in the previous section $ParA_\tau$ seeks to locally introduce parallelism and extend, i.e., grow, the code regions covered by parallel execution. It starts by internally parallelizing individual basic blocks, which consist of a sequence of instructions and do not impose any difficulties due to complex and potentially irregular control-flow.

$ParA_\tau$ first uses the dependence analysis to derive a dependence graph of the instructions of the basic block. Note that this graph is directed and acyclic as the instructions do not contain any loops. We call this graph the dependence DAG (*DepDAG*).

The *DepDAG* is then used to formulate the parallel scheduling of the instructions as an integer linear optimization problem and uses an ILP solver [3] to come up with an optimal schedule which minimizes the critical path execution time. Note that optimal here naturally means optimal with respect to the cost function and the chosen cost model. This can only be an approximation of real execution time, which in turn is statically unpredictable and input dependent in general.

A detailed description of the resulting schedules and in particular the *ILP* formulation is left for Chapter 6 in which all details are given. While the ILP used by $ParA_\tau$ is way simpler than the one used by $ParA_\gamma$ and described in Chapter 6, it shares the same basic ideas and in fact could be replaced by that version. What is important to note here is that individual basic blocks are parallelized to form individual sections of parallel execution. We call those sections parallel sections, or *ParSec*s. Those sections are then extended by pulling in surrounding code and uniting them with other parallel sections. This extension, or *ParSec-Propagation*, is described in Subsection 4.1.4.

*ParSec*s are manifested in parallel control-flow graphs (*ParCFG*s), which are produced as the result of the compile-time parallelization part of $ParA_\tau$ within the *Sambamba*

[3] We use the IBM Cplex ILP solver in our current implementation. This choice is nevertheless not important for our approach and the solver can easily be replaced by another implementation.

framework. The *Parallel Control-flow Graph* and its constituents that distinguish it from a regular *CFG* are described in the next section.

4.1.3 Parallel Control-flow Graph (ParCFG)

```
1 #pragma omp parallel for
2 for(int idx = 1; idx < N; ++idx) {
3   norm[idx] = vals[idx] / norm(vals, N);
4 }
```

Adding parallelism to the compiler IR, and thus making it explicit to the compiler, is important to enable optimizations in the presence of parallelism. Imagine, for instance, the *OpenMP* parallel for loop to the right. Such a construct is typically transformed by the compiler frontend and completely invisible to the optimizing middle and backend. Typically, the body of the parallel loop is externalized to a separate function, and the actual loop replaced by a call to the corresponding runtime system. This is called early proceduralization and nearly prohibits all important compiler optimizations, like for instance loop invariant code motion.

But full integration of parallelism is not only a matter of optimizations, and thus performance. It is also a matter of correctness. In order to keep the engineering overhead as low as possible it has been frequently proposed to integrate parallelism via early proceduralization, as mentioned above, or by adding meta data or compiler intrinsics to the IR which marks regions of possible parallel execution, but might be ignored by optimizations not aware of the semantics of it. The main motivation of such proposals has been to minimize the necessity to adapt every optimization in order to make it aware of the parallelism, which would be invasive and error-prone. Unfortunately, parallelism *is* a very invasive concept, as can be seen in Figure 4.7.

Figure 4.7a shows a code region with parallel tasks marked by compiler intrinsics *parallel.task.start(<id>)* and *parallel.task.end(<id>)*. Whether we choose intrinsics, meta data or similar minimally invasive constructs to mark parallelism is of minor technical importance. Important however is that an optimization like *common subexpression elimination* (*CSE*), unaware of the parallel semantics, is allowed to produce the result shown in Figure 4.7b. In the example, it is allowed to do so because the standard dominance information, on which many optimizations rely, is not correct: despite the fact that the

```
1 parallel.task.start(1)
2
3   some_fun(n/2)
4 parallel.task.end(1)
5 ...
6 parallel.task.start(2)
7   other_fun(n/2)
8 parallel.task.end(2)
```

(A)

```
1 parallel.task.start(1)
2   tmp = n/2
3   some_fun(tmp)
4 parallel.task.end(1)
5 ...
6 parallel.task.start(2)
7   other_fun(tmp)
8 parallel.task.end(2)
```

(B)

FIGURE 4.7: Illegal transformation due to insufficient integration of parallelism into the compiler IR.

two program parts which are supposed to run in parallel are necessarily written down in a sequential order, the implicit parallel semantics dictates that none of both regions precedes the other. Consequently, none of both regions dominates the other and moving a commonly used subexpression into either of both regions, as done in Figure 4.7b produces a wrong result.

The parallel control-flow graph (*ParCFG*) resembles the structure of the *CFG* and adds constructs of parallel execution. This means that we distinguish between sequential and parallel control flow and corresponding edges. Parallelism is introduced into the *ParCFG* by fork instructions, called π_s. π_s instructions form the entrance to a parallel section which in turn is terminated by a join instruction π_e. A parallel section consists of multiple parallel tasks, which contain parts of the code that can potentially be executed in parallel.

The control-flow edges originating in a π_s represent parallel control-flow and end in the first basic block, the so-called head, of a parallel task. A π_s is typically succeeded by many parallel tasks of its containing parallel section. It has exactly as many outgoing edges as the containing section tasks.

Parallel tasks in their basic form are single entry single exit regions (called SESE-Regions or Hammocks in the literature). They are terminated by the one unique π_e instruction of their containing parallel section. The π_e joins, or synchronizes, parallel execution.

Figure 4.8 shows the *ParCFG* of the *performTask* method as it is produced by the static part of $ParA_\tau$'s parallelization.

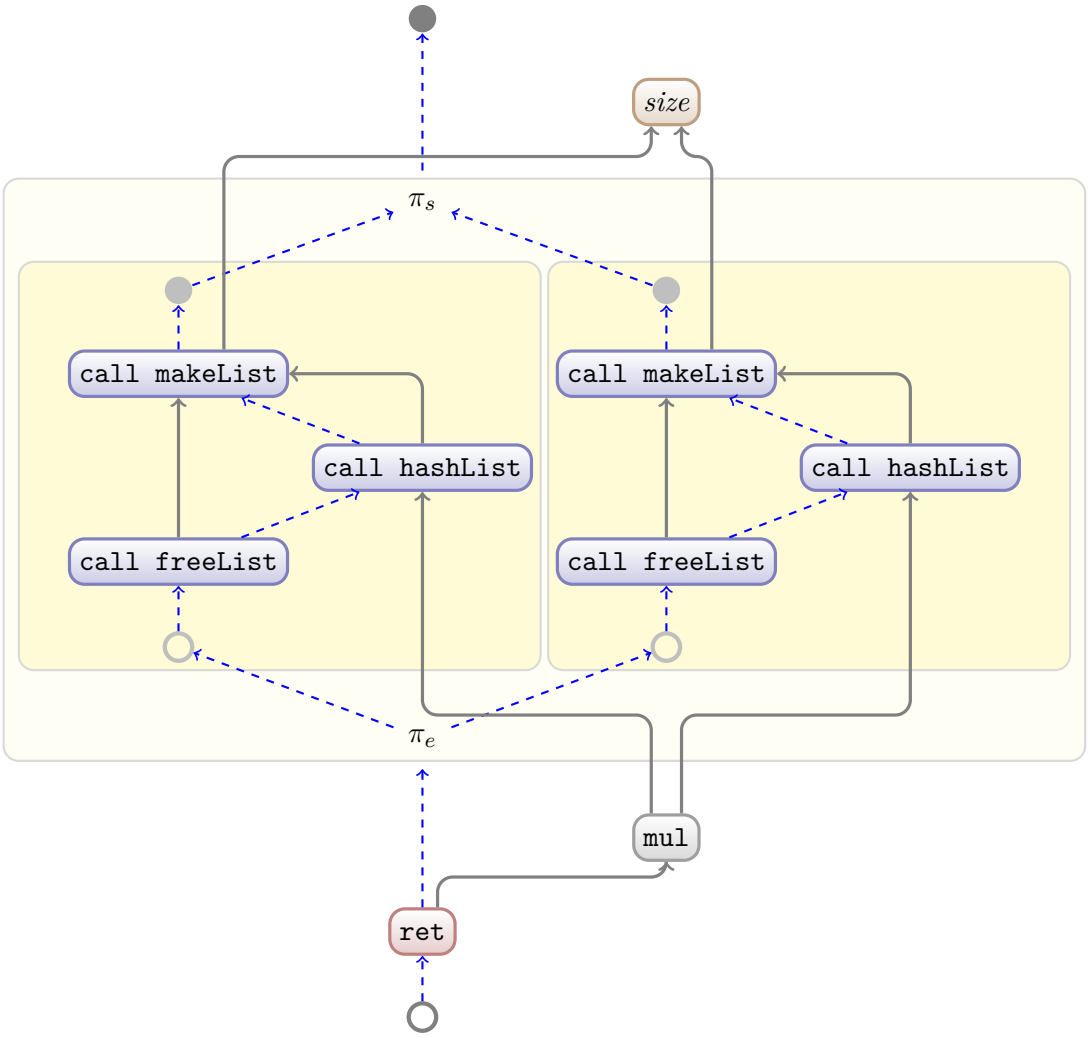


FIGURE 4.8: ParCFG for parallel version P_0 of the *performTask* method as automatically derived by *Sambamba*. The outer box depicts a so-called parallel region consisting of transactions depicted by the inner boxes. Each parallel region is entered via at least one π_s node and left via the one π_e, which is unique per region. A π_s forks parallel execution and π_e joins again after all contained transactions completed.

Parallel control-flow graphs do form an intermediate representation which makes the static parallelism of a function explicit. They are independent of the form of execution, being it parallel or sequential, chosen later on during code generation. Should parallel code be generated, a π_s would result in code which produces the necessary communication and synchronization primitives at runtime and starts parallel execution. The corresponding π_e would be replaced by some form of barrier or synchronization. A simple and straight forward possibility would be to produce a cilk [41] (or Cilk+ [42]) style program: each parallel task t of a parallel section is extracted into a function f_t with parameters corresponding

to values used by but not produced within t. The π_s is replaced by a number of *spawn* instructions potentially spawning all tasks of the section, the π_e is replaced by a *sync* instruction. It would be as easy to produce an OpenMP [6, 7] or Intel TBB [40] based parallel version, as is to linearize the tasks to produce a sequential version.

The *ParCFG* forms an essential building block to the modularity and extensibility of *Sambamba*, which provides the infrastructure for final code generation. Parallelization modules based on *Sambamba* (like $ParA_\tau$ and $ParA_\gamma$) can concentrate on finding the parallelism inherent in the application and to produce *ParCFG*s where appropriate. *Sambamba* provides the infrastructure to generate the parallel code.

Furthermore, $ParA_\tau$ only profits from a subset of the features of the *ParCFG*. As Chapter 5 describes, the simple fork/join based parallelism of the *ParCFG* is sufficient to encode a large range of different forms of parallelism, including many loop parallelization schemes. Recently, and supporting our own efforts, Schardl et al. [96] (also [97]) proposed an extension of the *LLVM* IR named *TAPIR* which is based on three constructs introducing parallelism into the IR: *detach*, *reattach* and *sync*, which are of a similar expressive power than the *ParCFG* parallel sections with their π_s and π_e constructs. The findings of this thesis motivated and influenced a group around Johannes Doerfert at Saarland University who is working together with the authors of TAPIR at *MIT* to introduce proper parallelism constructs, including a well-defined parallel semantics, to the LLVM IR. An important feature of this new IR, which the *ParCFG* does not fulfill, is the full compatibility to existing analyses and transformations, while preserving correctness of analyses which are unaware of the parallel semantics.

4.1.4 Parallel Section Propagation

By applying the scheme described in the previous sections, in particular basic-block-wise parallelization, $ParA_\tau$ is able to execute calls, for instance, which are contained in the same basic block, in parallel to each other and by that to potentially gain some promising performance improvements. This however seems to be severely restricted and is only slightly more powerful than the first phase of parallelization as described by Rugina and Rinard [56], in which all call instructions are marked with a cilk *spawn* and followed by

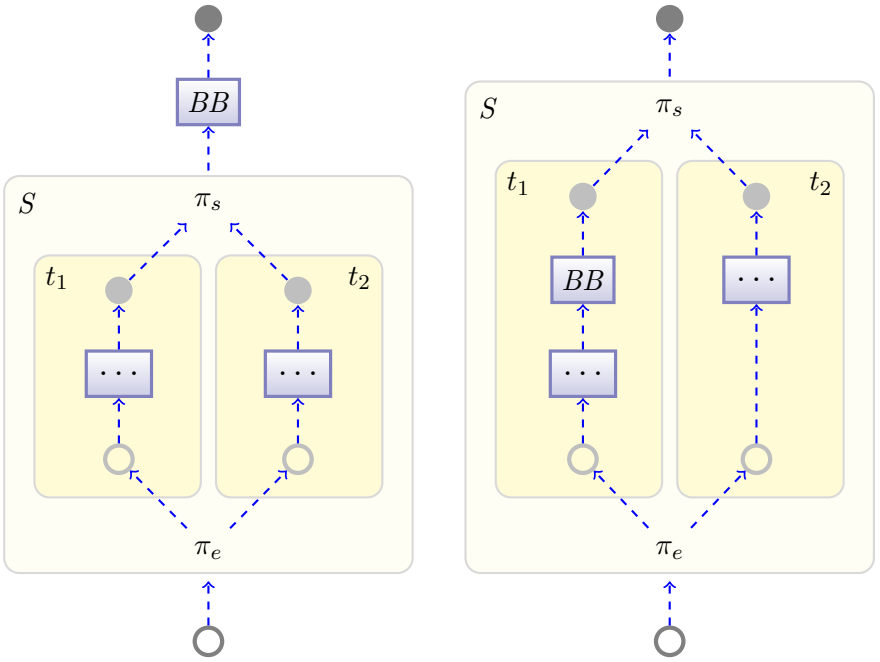
an immediate cilk *sync*. It is more powerful in the sense, that this step is already able to effectively parallelize close-by call instructions and surrounding code sharing the same control conditions, thus being typically placed in the same basic block. While this is a severe restriction, it is already sufficient to parallelize the *performTask* method above or the benchmarks used in [56].

In order to enable $ParA_\tau$ to parallelize across basic block boundaries however, we have to extend formed parallel sections, which $ParA_\tau$ does by pulling in the surrounding code into the parallel sections. It effectively extends, or propagates, the parallel sections, similar to moving the *sync* statement in [56]. We explain parallel section propagation as it is important to $ParA_\tau$. For $ParA_\gamma$, the final parallelization approach developed in the course of this thesis, however it is of minor importance. We therefore keep the description on an intuitive level.

At first we need a few definitions: effect sets $Eff_r[t]$ and $Eff_w[t]$ ($Eff_r[S]$ and $Eff_w[S]$) as described earlier in Subsection 4.1.1 are intuitively defined for parallel tasks and parallel sections of the *ParCFG* as the union of the respective effect sets of contained instructions. $nme_r[S]$, $nme_w[S]$, and $term[S]$ ($nme_r[t]$, $nme_w[t]$, and $term[t]$) are similarly defined as the disjunction of the respective values of contained instructions. Based on these effect definitions, $conf[\cdot,\cdot]$ and $(\cdot \rightarrow \cdot)$ are also defined for basic blocks, parallel tasks, and sections.

Furthermore, a statically estimated execution cost (execution time) $\|t\|$ is defined for a task t (similarly $\|B\|$ for a basic block B), which for this simple approach is based on the number of contained instructions (including the instructions contained in transitively called and statically resolvable functions) each multiplied by a fixed constant for potentially surrounding loops.

Consider the scenario shown in Figure 4.9a. It shows a parallel section S, containing two tasks t_1 and t_2. The section is preceded by a basic block BB which should be pulled into the section. Figure 4.9b shows the outcome of propagation, which however is only one of many possibilities:



(A) Before propagation. (B) After propagation.

FIGURE 4.9: Parallel section propagation scenario (A) with one possible outcome (B).

BB $\rightarrow t_1$**.** If $conf[BB, t_1] \wedge \neg conf[BB, t_2]$, i.e., BB conflicts with t_1 but not with t_2 , it is moved into t_1. This is the scenario shown in Figure 4.9b.

BB $\rightarrow t_2$**.** Similarly, if $\neg conf[BB, t_1] \wedge conf[BB, t_2]$ BB is moved into t_2.

BB $\rightarrow t_1 \wedge$ ***BB*** $\rightarrow t_2$**.** If $conf[BB, t_1] \wedge conf[BB, t_2] \wedge (Eff_w[BB] = \varnothing) \wedge \neg nme_w[BB] \wedge \neg term[BB]$, BB can safely be duplicated in favor of extending the region covered by parallel execution. Also, duplication is only an option if $\|BB\|$ is below a configurable threshold.

BB $\rightarrow t_3$**.** If $\neg conf[BB, S]$, BB is put into a new parallel task t_3.

BB $\not\rightarrow$ ***S*****.** In case BB conflicts with more than one task and has itself an observable effect or is too large, which both prohibits duplication, propagation stops.

The mentioned scenarios exhaustively cover only the simple case of a single block preceding a parallel section and having itself only a single successor. They should however give a feeling of what parallel section propagation is and how it partially removes the limitations of the basic-block-based parallelization of $ParA_\tau$. The implemented propagation algorithm

covers more complex cases and indeed is also able to propagate along complex control flow. In theory, it can extend parallel execution to cover whole functions this way. In practice however, the approach is limited in its capabilities to propagate parallel regions across blocks with excessive memory or non-memory side effects, because the scheme is unable to reorder code. For this reason, parallel section propagation as a central aspect of parallelization is superseded in $ParA_\gamma$ by the simpler and practically more powerful *PDG*-based whole function parallelization.

Parallel section propagation however is used as an optimization step after the PDG-based parallelization of $ParA_\gamma$. It is able to improve on the results achieved by $ParA_\gamma$ or semi-automatic parallelization (see Chapter 10) by performing local optimizations, which $ParA_\gamma$ is unable to do, as it is by definition limited to transformations among nodes sharing the same control-conditions, and which are typically too fine-granular to be dealt with by a manually parallelizing programmer.

4.1.5 Load-based Adaptive Dispatch

The *ParCFG* is used to compile a final parallel version for each parallelizable function of the program. Those parallel versions are then packed into the binary together with their sequential counterparts. It is left to the runtime system to decide which version to execute upon a call to the respective function. $ParA_\tau$'s runtime system decides, or dispatches, based on the system load: if all resources of the surrounding system are already fully loaded, then there is no use in further parallel execution, which would do nothing but to introduce overhead and to over-subscribe the system.

Unfortunately, getting the system load inevitably requires to call to the operating system, which is expensive and therefore cannot be done upon every single call to a parallelized function with the sole purpose of deciding if the parallel version should be executed or not. In order to be able to compensate for the overhead of the dynamic dispatch mechanism and to increase the profitability even for small parallelized functions it needs to be as lightweight as possible. The overhead of getting the CPU load each time might very well outweigh the execution time of the function to call. Therefore, a separate thread polls the

cpu load in a configurable interval, by default once every second, and stores it in globally shared memory for the dispatch mechanism to read once per call.

Section 7.4.1 gives more details on load based dispatch; Section 7.4 further puts it in reference to different alternatives implemented in $ParA_\gamma$.

4.1.6 Lessons learned from $ParA_\tau$

The simple basic-block-based parallelization scheme of $ParA_\tau$ has to be considered a test-bed and development environment for the techniques and intermediate representations described in this section: *DSA*-based memory effect computation, the *ParCFG* and adaptive dispatch. All of those are part of $ParA_\gamma$ in a more or less extended and improved version.

Apart from these purely didactic reasons, $ParA_\tau$ is a fully working task based parallelization scheme worth mentioning. In practice, it is limited by its propagation along the more or less programmer dictated execution path and its inability to propagate "around" blocks that are not suitable for parallel execution or where the conservative memory analysis has to assume significant memory effects that prohibit parallel execution.

The knowledge gained during development of $ParA_\tau$ lead to using the program dependence graph (*PDG*) [28] as the main intermediate representation of $ParA_\gamma$ to overcome these limitations. The *PDG* by design does not contain any control-flow as it is dictated more or less arbitrarily by the program order. It is solely based on control and data dependences, but sufficiently encodes the program semantics to be able to synthesize a regular control flow graph by *sequentializing* the *PDG* [98]. It is the perfect representation for parallelization as already noted by Sarkar [54].

4.2 Speculation Support

Although $ParA_\tau$ does not make use of speculation, it is an important technique that enables automatic parallelization. Recent work of Niall et al. [11] goes as far as claiming that reasonable parallelization cannot solely rely on static dependence analysis and instead has to

use speculation to fully exploit the only dynamically exploitable parallelism. Unfortunately, existing techniques and speculation mechanisms come at a high price in terms of runtime overhead, which needs to be reflected in the decisions of an automatic parallelizer. The problem with speculation is that its overhead inherently depends on the misspeculation rate, which in turn depends on runtime features: The number of threads/tasks running in parallel as well as the structure of the input.

Developing a framework for speculative execution is not part of this thesis' work and therefore not explained in thorough detail. Instead, an overview is given over the different options implemented in *Sambamba* as part of a different PhD thesis [29], as far as it is useful to put decisions explained in later chapters of this thesis into context.

Two different speculation approaches have been developed and integrated into the *Sambamba* framework. One approach is based on Software Transactional Memory and implemented as an extension of TinySTM [99, 100].

The other approach [14] is based on a concept commonly known as thread-level speculation (TLS). It has been specifically implemented from the ground up to drive speculative execution as required by the parallelization framework described in this thesis.

4.2.1 Software Transactional Memory

Transactional memory systems [101] are motivated by the corresponding concept in database systems and typically guarantee atomicity and isolation. Atomicity refers to the property that the (memory-)effects of instructions contained within the same transaction are, from an external point of view, all visible (committed to main memory) at once or not at all. The system guarantees that the effects of a transaction are never partially visible only. We further differentiate between strong and weak atomicity, of which the latter guarantees atomicity only between different transactions; the former also guarantees atomicity between transactions and the surrounding code outside of the control of the transactional memory system.

Transactional memory systems implemented in software only (*STM*) typically guarantee weak atomicity as they rely on instrumenting the code contained in transactional sections.

Guaranteeing strong atomicity would require the system to instrument the whole code, including dynamically linked parts, which poses technical issues, but also comes with the corresponding non-negligible overhead.

Hardware transactional memory systems (*HTM*) in contrast typically provide strong atomicity. The usual implementations, like the *Intel Transactional Synchronization Extensions* (*TSX*) [102], are based on the cache coherence protocol and impose significantly less runtime overhead in comparison to software only implementations.

One of two speculation systems implemented as part of the *Sambamba* framework is based on TinySTM [99, 100] and comes with the typical performance overhead of an *STM* system. To make it usable in an automatic parallelization context, where keeping the sequential semantics of the parallelized application is of importance, the implementation contained in *Sambamba* adds a commit order: the order between transactions resulting from automatic parallelization (for instance as done by $ParA_{\gamma}$) is defined by the broken, i.e., speculatively ignored, dependences. This is an important criterion that heavily influences parallelization decisions. As a result of this requirement it is illegal to form transactions in *Sambamba* whose speculatively ignored dependences impose a circular commit order between the transactions. Chapter 6 describes in detail how this is guaranteed by $ParA_{\gamma}$.

Due to its typically small setup overhead per transaction (in contrast to the above mentioned high overhead per protected memory operation and commit) *STM* is particularly well suited to replace locking primitives protecting comparably small critical parts of big parallel tasks. It is not, however, a good fit for completely protecting very big speculative parallel tasks. To cover that use-case, *Sambamba* additionally provides an alternative speculation mechanism based on process forking as described in the next section.

4.2.2 K-TLS

K-TLS is in most cases the speculation system of choice in *Sambamba*. It is a so-called *Thread-level speculation system* (*TLS*) based on process forking to isolate the memory effects of individual transactions to guarantee atomicity and isolation. In contrast to the *STM* implementation described in the previous section, *K-TLS* and similar systems come

with a high initial setup overhead per speculatively spawned task, but nearly no overhead per protected memory operation within a task. Upon completion of a speculatively parallel task and a successful conflict check, the memory effects of a transaction are made accessible to the main process by atomically moving over written memory pages.

The *K* in *K-TLS* comes from *kernel* and hints at the implementation as part of the operating system kernel. Only this way it can effectively use the hardware based memory management to keep the overhead as low as possible. Another advantage is that this way, speculatively ignoring possible system calls is straight forward, even for dynamically loaded binaries not allowing for instrumentation of the code. All system calls are handled, and can therefore be intercepted, by the OS kernel. If a system call happens speculatively, the transaction can be aborted or stalled until completion of transactions preceding the one executing the system call in the commit order, which *K-TLS* requires just like the alternatively usable *STM*.

The high setup cost of transactions in the *K-TLS* is a bearable cost given the low overhead of memory operations, as big transactions are typically the goal of task based parallelization of general purpose applications. The downside of *K-TLS*, or more generally systems exploiting the virtual memory system for conflict detection, is the granularity of conflict detection, which typically is on the page-level. That means that a conflict is detected, and consequently the speculative execution rolled back and repeated, if two speculative tasks write to the same memory page of typically four kilobytes. This granularity might, depending on the application, lead to a significant amount of so-called false conflicts caused by two tasks writing to completely disjoint regions of the same memory page.

The *STM* system of the previous section is instead able to detect conflicts on the word level which nearly eliminates the risk of false conflicts but comes, due to the necessary instrumentation of the code, with the limitations and draw-backs, especially performance-wise, described earlier.

K-TLS+ is a hybrid system that seeks to overcome the limitations induced by the page-level conflict detection by again relying on instrumentation of speculatively executed code to resolve potential false conflicts within a page. The granularity, and with it the overhead of the required instrumentation, is configurable in *K-TLS+*.

4.3 The Dynamic Nature of *Sambamba*

Choosing the parts of a program to profitably parallelize, the right speculation system, the right parameters for a chosen system, or dropping speculatively parallel execution completely, is a decision that is only reasonably made based on dynamically collected characteristics of parallel execution, which not only depends on the program itself but also on user input and the execution environment. The frequency of (false) conflicts cannot be statically anticipated as it heavily depends on the parallelism available, which in turn depends on the user input and the number of available computational resources like CPU cores. The overhead caused by re-execution of failed transactions of course also depends on the input.

Sambamba therefore provides all the infrastructure to leave the decision on what, where and how to parallelize to a runtime system. This is what $ParA_\gamma$ makes heavy use of and why it does exactly that: statically finding promising parallelization candidates based on the information of earlier runs of the application or static estimates of important parameters and leaving the final decisions and tuning to the runtime system, also implemented on top of *Sambamba*. Just-in-time compilation allows to completely reassess major decisions of parallelization without overly instrumenting the application to allow for dynamic tuning. Dynamic dispatch mechanisms, of which two more are described in Section 7.4, allow for low overhead dynamic tuning of parallel execution and reduction of parallelism-induced overhead on system oversubscription. Finally, the work-stealing dynamic scheduler of Intel TBB [40], on which the *Sambamba* runtime system relies for parallel execution, takes care for even and cache friendly distribution of parallel work. All mechanisms combined result in a highly flexible and adaptable form of parallel execution.

In this chapter, you have seen a conceptual overview of the *Sambamba* framework, an extensible static/dynamic compilation and runtime environment based on the LLVM compiler infrastructure.
$ParA_\tau$, a first task-based parallelization scheme implemented on top of *Sambamba* has been presented and used to introduce important techniques and terminology, which

also form the basis of $ParA_{\gamma}$, our final approach for generalized task parallelism. These techniques include in particular a context-sensitive and inter-procedural memory access analysis, the parallel control flow graph (*ParCFG*), and runtime adaptive dispatching based on the current system load.

Chapter 5

Generalized Task Parallelism — *ParA*$_\gamma$

The goal of our parallelizer *ParA*$_\gamma$, which this and the following Chapters 6 and 7 will introduce, is to find for each function of an application a set of *parallelization opportunities*. From these candidates *ParA*$_\gamma$ chooses the combination that best fits the execution environment at runtime. Parallelization opportunities are found in the form of arbitrary, possibly nested, regions of code amenable for parallel execution.

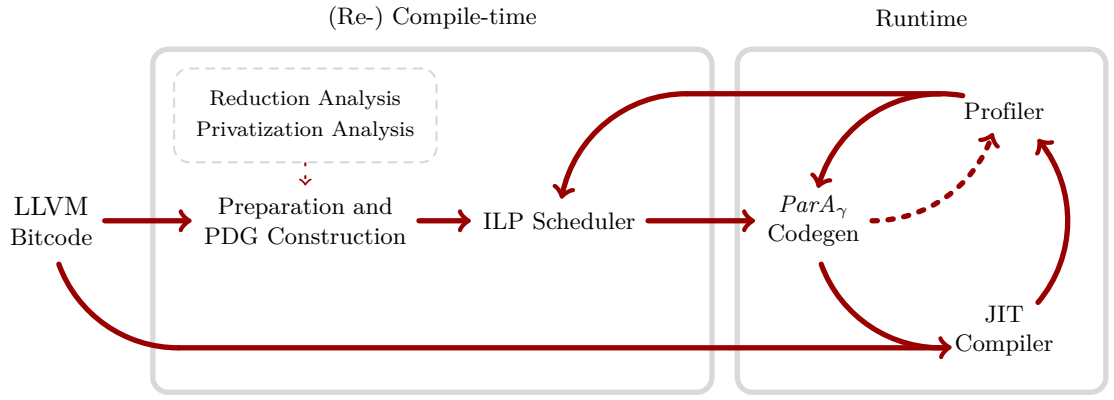


Figure 5.1: Overview of the *ParA*$_\gamma$ parallelization system.

$ParA_\gamma$ consists of two parts: a *compile-time component*, performing most of the time-consuming program analyses, transformations and scheduling offline; and a *runtime component*, building on statically gathered information and continuously collected runtime profiles to perform online adaptive optimizations.

Figure 5.1 gives an overview of the workflow of $ParA_\gamma$. The application, in the form of its compiled LLVM bitcode, is read as input. This enables $ParA_\gamma$ to deal with programs written in different languages; no syntactic information is required. The resulting control flow graph of each individual function is then preprocessed and a program dependence graph (PDG) constructed. Also, reduction and privatization opportunities are identified and reflected correspondingly in the PDG. A scheduler based on integer linear programming (ILP) is used to find a set of parallelization candidates per function; it takes into account statically estimated and, if available, dynamically gathered profiling data and generates an optimal schedule with respect to the execution cost-model, expressed in its constraints. The found candidates are called local parallelization candidates and reflect parallelization opportunities which are statically deemed beneficial; the decision if a local candidate will be instantiated is left to the runtime system. Note that the set of candidates may contain, but is in no way limited to, parallel loops. It may well be that the scheduler decides to execute arbitrary regions of code in parallel to each other. The cost-model explicitly reflects the cost of exploiting reduction or privatization candidates. It is up to the scheduler to decide if and where such opportunities are worthwhile to realize with respect to its optimization function.

At runtime, the statically found parallelization candidates are evaluated, considering the actual execution environment; one parallel version of each function is generated for the best combination of its local parallelization candidates. To decide on the quality of a combination, a modified version of the scheduler cost function is used. Using a just-in-time compiler, this parallel version is compiled and patched into the running application. A dynamic dispatch mechanism is installed to decide, upon calls to the function, whether execution should proceed with the parallel or the sequential version. The application is continuously monitored by an efficient, sampling based profiler. The empirically gathered execution time of individual call sites allow $ParA_\gamma$ to react to changing runtime conditions.

$ParA_\gamma$ works in a fully automatic way and is used like a regular C/C++ compiler. Additionally, speculation hints can be given by the programmer to guide parallelization. Nevertheless, $ParA_\gamma$ in no way relies on the existence of such hints, nor on their accuracy.

In the following, this chapter will lay the foundation of generalized task parallelism by first introducing $ParA_\gamma$'s flavor of the program dependence graph (PDG) as its central program representation. It will furthermore introduce important parallelization enabling techniques before the following Chapter 6 will go into the details of scheduling for parallelism. While Chapters 5 and 6 focus on the static parts of $ParA_\gamma$, Chapter 7 provides details on the dynamic capabilities of $ParA_\gamma$'s runtime system.

5.1 Program Representation

$ParA_\gamma$ works solely on program dependence graphs (PDG) [28]. At compile-time a PDG is constructed for each function. Such a PDG is kept during compilation and, if parallelism has been found, also during application runtime. The following sections explain in detail how the $ParA_\gamma$ PDG looks like and what form of extended information is stored in it.

5.1.1 Program Dependence Graph (PDG)

As stated by Sarkar [54], the PDG is a perfect representation to express and analyze parallelism: it abstracts from overly restrictive implementation-dictated execution order and unveils all available parallelism by ordering instructions solely based on actual dependences. The challenge is to find the *right granularity of parallel execution*, as the parallelism reflected by the PDG is too fine grained in general. Computation nodes need to be grouped to form coarser parallel tasks, costly enough to outweigh the overhead of packing and spawning.

In $ParA_\gamma$, individual PDG nodes represent basic blocks of instructions. Before computing the PDG, basic blocks are split to isolate instructions of interest and increase the freedom to schedule them independently. Such instructions are, for example, accesses to reduction

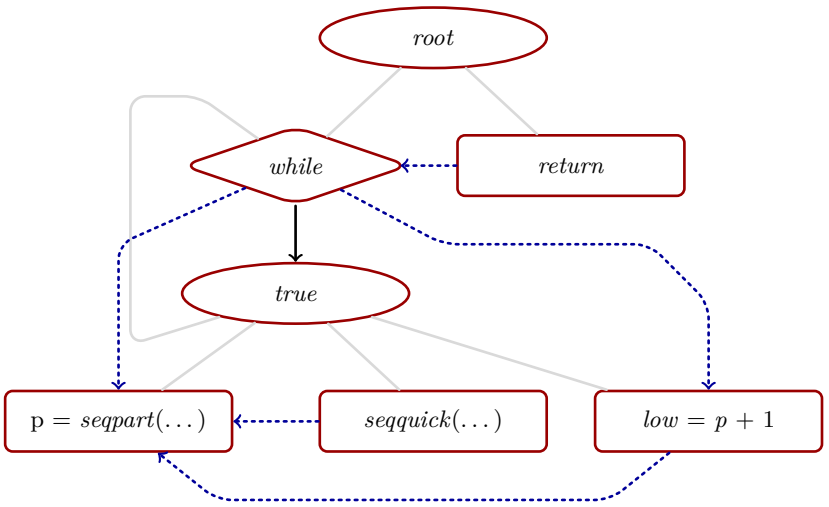


FIGURE 5.2: The simplified PDG of *seqquick*: data dependences are depicted by dashed arrows, control dependences by solid arrows. Light solid lines depict parent relationship. Note how loops are represented in the PDG.

and induction variables as well as function calls.[1] Details on the splitting of basic blocks prior to PDG construction are given in Subsection 8.2.1.

ParA$_\gamma$ computes data dependences by an interprocedural, context-sensitive points-to analysis based on the data structure analysis (*DSA*) [23] as described in Subsection 4.1.1. In addition to the analysis described earlier, *ParA*$_\gamma$ adds an array access analysis based on the polyhedral toolchain of *Polly* [103], the polyhedral optimizer of the LLVM framework, to disambiguate array accesses in loops[2]. Furthermore, *ParA*$_\gamma$ allows for user annotations to give hints to the dependence analysis. Currently, such hints are useful in the presence of recursive functions, for which DSA severely over-approximates by unifying the effects of all functions involved in the strongly connected component in the call-graph.

As an example, the PDG of the *seqquick* function of Figure 1.4c is shown in Figure 5.2. The PDG contains nodes (N) partitioned into sets of three different types:

- Regular nodes (R), depicted as boxes, represent simple instructions or basic blocks.
- Decision nodes (D), depicted as diamonds, represent basic blocks with more than one successor in the control flow graph. In the LLVM context these are basic blocks

[1] Note that the restriction to the basic block level is not a limitation of the approach but instead a mere technical trade-off between processing time and transformation freedom.

[2] The array dependence analysis of *ParA*$_\gamma$ has been implemented by Johannes Doerfert, who also contributes to *Polly*.

terminated by conditional branches, switches or possibly exception throwing calls (*invokes*).

- Finally, group nodes (G), depicted as ovals, group possibly multiple nodes (called its *children*) sharing the same control condition. Each group node—except for the unique root node—is directly control-dependent on exactly one decision node in the PDG.

Each group node represents a control condition, which is the conjunction of all conditions of decision nodes on the path from the designated PDG *root* node to the corresponding group. Once this condition is fulfilled, all its child nodes are to be scheduled for execution. Only the data dependences between the subgraphs reachable from the group node's children restrict parallel execution. Within one group node, no complex control flow has to be taken into account: a property that makes the group nodes particularly interesting in the context of synchronous task parallelization.

In the remainder of this thesis we make frequent use of the following important terminology: we call the PDG sub-graph without data dependences, i.e., with control-dependences and parent relationships (induced by group nodes, see Figure 5.2) only the *control-dependence sub-graph* of the PDG. Furthermore, by *reachable PDG sub-graph, rooted in node* n we mean all PDG nodes, which are (transitively) control-dependent on n.

The purpose of the scheduler as described in Chapter 6 is to find for each group node in the PDG a schedule of its respective children, representing the whole subgraph reachable from the particular child node. Note that in this way it is possible, even natural, to generate nested parallelism. It is up to the runtime component of $ParA_\gamma$ to decide at which group nodes, and consequently also at which nesting levels, to make use of the parallelization opportunities provided by the static scheduler.

5.1.2 Sequentialization of the Program Dependence Graph

As stated earlier, the program dependence graph is the perfect representation of parallel programs. It removes the *structurally imposed program execution order* and leaves only the

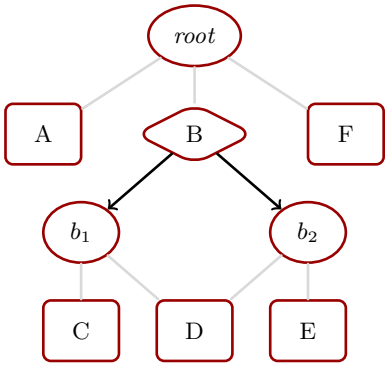


FIGURE 5.3: A simple PDG for sequentialization.

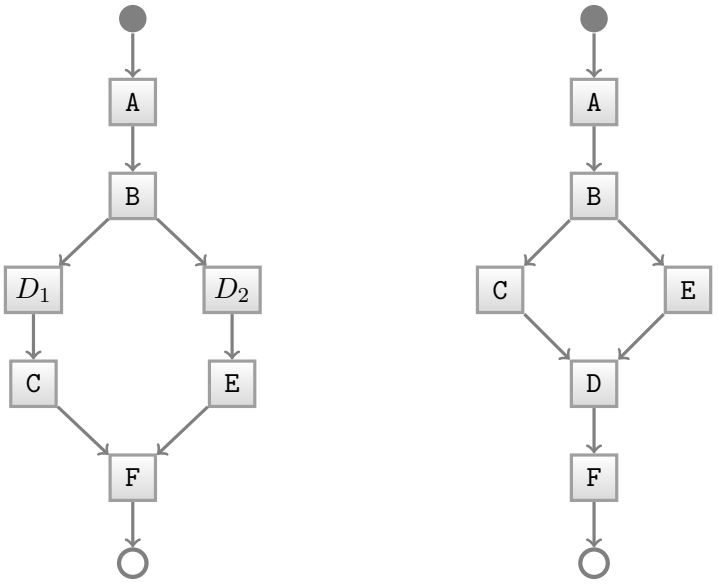
strictly necessary control and data dependences which are to be respected to guarantee preservation of the sequential program semantics. In order to be executed by the machine, the program's statements however need to be put in an execution order in the form of a control flow graph. This process is called *sequentialization of the PDG* and has been the subject of extensive research with the goal of solving the issue for different forms of programs ranging from loop-less programs [104–106] to programs containing only single-entry loops [107] to irreducible programs containing arbitrary loops [98] respectively. The more recent work of Zeng et al. [108] has dealt with the efficiency of the code sequentialized from a PDG with possible interleaving, i.e., circular dependences between disjoint PDG sub-graphs. The goal to achieve in the optimal sequentialization is minimal duplication of code, or minimal number of guards[3].

As an example consider the simple PDG shown in Figure 5.3[4], and the two possible sequentializations in Figures 5.4a and 5.4b, the latter being optimal, while the former required duplication of node D due to a sub-optimal order of generating code for the children of group nodes b_1 and b_2.

Steensgaard [98] shows how to optimally sequentialize PDGs even for irregular code based on the so-called *external edge condition* (*EEC*). Our situation is special, however, and in fact allows for a simpler solution: the PDG used in *Sambamba*/$ParA_\gamma$ is computed from an existing control-flow graph, and mostly used for analysis purposes only. Since $ParA_\gamma$ does not introduce new dependences into the PDG it is clear that a duplication and

[3]Sequentialization can always be duplication-free provided enough predicates are inserted into the code guarding the execution.

[4]For the sake of a simple example please ignore the fact, that D is not control-dependent on B.



(A) Non-optimal with D duplicated. (B) Optimal.

FIGURE 5.4: Sequentialization of the PDG shown in Figure 5.3.

guard-free sequentialization of the PDG exists: the original CFG. In the general case this is not always true (see Ball and Horwitz [107]). Furthermore, and more importantly, we can keep the information on this sequentialization, at least implicitly, by storing for each PDG node a group-unique ID based on the post-order numbering of the blocks in the control flow graph with loop closing edges removed. IDs are group-unique, i.e., unique among the children of each PDG group node, instead of unique for the whole PDG, since group nodes themselves do not have a correspondence in the CFG, and therefore no corresponding ID. Instead, group nodes inherit the ID from the decision nodes they belong to. The basic idea then is that during sequentialization of a PDG group node, the children are ordered in descending order of their respective IDs, which for a loop-less program results in the original, duplication-free CFG.

Loops impose a different situation: without loops, the children of the reachable PDG sub-graph, rooted in node n always have a smaller ID than n itself, following from the definition of control-dependence, which requires reachability in the CFG, and post-order numbering, which guarantees that all nodes reachable from n are numbered before n. CFG loops however result in loops also in the control-dependence sub-graph of the PDG, which in turn result in nodes with higher IDs being reachable in the PDG. This needs to be

reflected when ordering the children of a group node g with post-order ID $poid(g)$ for sequentialization by sorting in two steps: first, all children c with $poid(c) < poid(g)$ are sorted in descending order of their IDs, followed by all children with $poid(c) \geq poid(g)$, also in descending order of their IDs. We call all children with $poid(c) < poid(g)$ *regular*, and all children with $poid(c) \geq poid(g)$ *loop-back*. The boundary between those two groups is called *loop-back boundary*, the one child with the smallest ID bigger or equal to that of its parent is called *reentrant* as it is the one closing the loop. Figure 5.5 illustrates the order of children and the terminology introduced above.

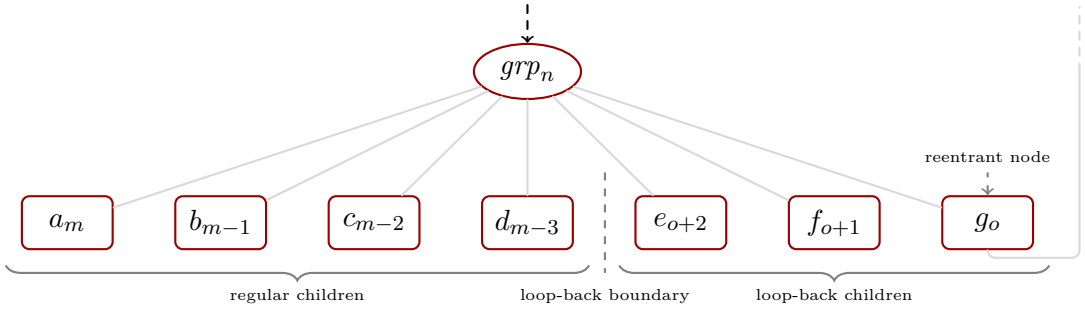
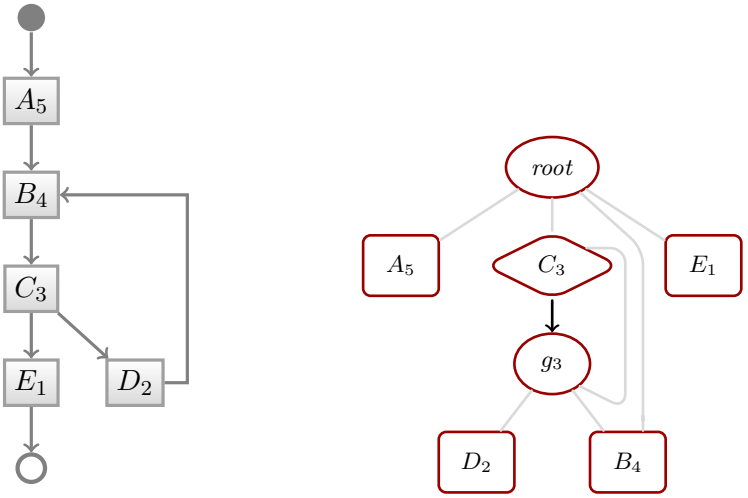


FIGURE 5.5: Order of children of a PDG group node grp with $poid(grp) = n$, and children a - g with subscripted CFG post-order IDs. $m < n$ and $o \geq n$.



(A) A CFG with post-order numbers of blocks as subscript.

(B) A PDG with CFG post-order numbers as node IDs.

FIGURE 5.6: A CFG and its corresponding PDG.

The loop-back boundary has a special meaning during parallelization: a schedule that executes all regular children of a group node in parallel to all loop-back children typically results in a DOALL parallelized loop. In cases of loop parallelization requiring realization

of a reduction or privatization, for instance, the loop-back boundary is the location to introduce fix-up code.

As a simple but complete example consider the CFG and its corresponding PDG in Figure 5.6, in particular the group node labeled g_3 with its three children D_2, B_4, and C_3, which have to be sequentialized in exactly this order to guarantee a minimal CFG. The *root* node's children have to be scheduled in the order A_5, B_4, C_3, E_1.

Storing the post-order ID as described above for each PDG node, especially since it has to be kept throughout the whole compilation process, within the binary as produced by *Sambamba*, and at runtime, might seem like a significant overhead. Indeed, for exactly that reason, we have implemented Steensgaard's algorithm at first to recompute and store the relevant ordering only in case it is needed. It turned out however that this particular information is frequently needed throughout the whole process. Additionally, we have been surely willing to trade memory for runtime efficiency. Finally, a unique ID per node is needed anyway for several implementation-related reasons.

5.1.2.1 Sequentialization of Parallelized PDGs

So far we talked about sequentializing a PDG with the goal to get the original, sequential, CFG without duplication of code or introducing execution guards. In case the children of a group node are scheduled for parallel execution, however, generating the *ParCFG* might require duplication of blocks, not only those directly contained in the parallel region, but especially those preceding it. Note that relying on the simple node ordering defined above still results in a minimum of duplicated code. Placing multiple copies of blocks however requires to select among the cloned values for later use by control-flow successors. This is being taken care of during *ParCFG* generation.

5.2 Parallelization Enabling Techniques

As mentioned earlier, parallelism enabling techniques like speculation, privatization and reduction are frequently considered strictly necessary to effectively exploit the parallelism

of general purpose applications (e.g., [11]). In this section, we will describe how $ParA_\gamma$ recognizes, models, and uses candidates for the above mentioned techniques in the context of generalized task parallelism, in particular abstracting from syntactical patterns or special code features.

5.2.1 Generalized Reduction

Reduction, in different shapes and flavors, has been frequently identified as an important parallelization enabling technique. Its importance is also reflected by the fact that many domain specific languages and tools dealing with parallelism contain reduction as a first-class citizen of the programmer's toolset. Most automatic parallelization techniques typically rely on reduction recognition and realization at some point in the toolchain. Often times this is done prior to the actual parallelization and, in addition to data privatization, used as a technique to resolve, or remove, data dependences before the actual scheduling for parallelism takes place. This way the induced cost can only play an indirect role in the decision for or against parallelization.

In the context of generalized task parallelization, where reduction also plays a major role, we seek to treat reduction and its realization using one of many possible ways as an integral part. The non-negligible implied cost of reduction realization that typically comes with the necessary use of atomic operations, privatization of reduction values or synchronization techniques as well as indirectly through the necessarily changed memory access patterns (and cache utilization) of the code, are modeled and taken into account in the choice of the right parallelization granularity and used techniques.

In the relevant literature, a reduction is typically defined on a syntactical level and described as code that more or less fits into the pattern $x = x \otimes exp$ where x does not appear in exp, and $\otimes$ is one of a few, typically hard-coded, reduction operations. Starting from that definition, this section develops and formalizes our notion of reduction and sets the used terminology. Furthermore, apart from the theoretical properties of a reduction operation, we define the practically implied costs and describe different ways of realizing a reduction. Chapter 6 describes in detail how reduction and its realization costs are modeled as part of the underlying optimization problem.

5.2.1.1 Syntactic Approach

Most existing approaches simply define a reduction operation at a syntactical level. The following basic definition is taken from Rauchwerger et al. [32] (Note that this is only their basic definition; we will discuss their extended definition in the following section.):

Definition 5.1 (Red_{stx_1}). A reduction variable is a variable whose value is used in one associative and commutative operation of the form $x = x \oplus exp$, where $\oplus$ is the associative and commutative operator and x does not occur in exp or anywhere else in the loop.

In our setting, we see several problems with an approach like that: first and foremost, as it is syntactically defined, reduction recognition needs to take place in the compiler frontend and is language specific. $ParA_\gamma$ on the other hand, is dealing with language independent *LLVM* Bitcode.

Second, a definition like Red_{stx_1} is very restrictive. There is no reason to reject code like the one shown in Figure 5.7 in which clearly the syntactical pattern is not met.

```
1  int x = 0;
2
3  do {
4    x += 23;
5    if (...)
6      x -= 12;
7    if (...)
8      continue;
9    x = x + some_pure_fun();
10 } while (...);
11
12 printf("The result is: %d%n", x);
```

FIGURE 5.7: Valid reduction on variable x.

Midkiff [30] goes in the same direction:

Definition 5.2 (Red_{stx_2}). Reductions Red_{stx_2} are defined as "operations that reduce the dimensionality of at least one input operation using a commutative reduction operation, $\oplus$". Further, to recognize such operations, "a compiler essentially looks for statements of the form $s = s \oplus expr$" where "first, the value of $expr$ must be the same regardless of the loop order it is evaluated in", and "second, the left-hand side s must not be used in other statements".

The limitations of the definition of Red_{stx_2} are essentially the same as for Red_{stx_1}, further talking only about reducing the dimensionality of one input operation (vector, array, ...).

5.2.1.2 Dependence-based Approach

The definition of Kennedy and Allen [8] is more to the point:

Definition 5.3 (Red_{dep}). A reduction has three essential properties:

1. It reduces the elements of some vector or array dimension down to one element.
2. Only the final result of the reduction is used later; use of an intermediate result voids the reduction.
3. There is no variation inside the intermediate accumulation; that is, the reduction operates on the vector and nothing else.

Again, the definition of Red_{dep} only talks about vectors and arrays (it was tailored towards FORTRAN 90 and available hardware reduction operations). Nevertheless, it abstracts from the syntactical appearance and relies solely on dependences (See Kennedy and Allen [8] for more details).

Still, a reduction as shown in Figure 5.7 would not be captured by Red_{dep}.

To deal with reduction operations potentially spread over multiple statements (as in Listing 5.7), Rauchwerger et al. extended their basic definition Red_{stx_1} as follows:

Definition 5.4 (Red_{ERS}). Instead of relying on the syntactical form as defined in Red_{stx_1}, the concept of expanded reduction statements (ERS) is introduced: an ERS is formed by following the def-use chains of variables used in the right hand side (RHS) of a potential reduction statement $x = y \oplus exp$. The goal is to verify that the reduction variable (x) is just passed through those variables (e.g., y) to the reduction statement. The source and sink of such a reduction chain define the ERS, which can then be validated according to the criteria in Red_{stx_1}.

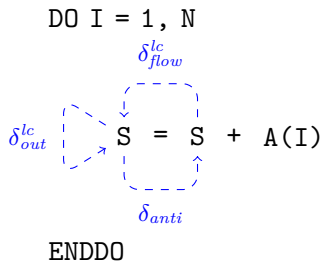


FIGURE 5.8: Dependences involved in a reduction operation (taken from Kennedy and Allen [8]). Dashed arrows depict data dependences.

Further they allow multiple reduction statements of the same form ($x = x \oplus exp$, or equivalent ERSs), provided all reductions over the same variable use compatible reduction operations (additive, multiplicative, ...). Also, control flow can be taken into account by allowing reduction variables to flow through γ statements (introduced by bringing the program into gated static single assignment (GSSA) form) in extended reduction statements.

In principle, a program as shown in Listing 5.7 would almost be recognized by the extended definition. Problems arise though with different forms of loops (while vs. do) or early loop exits (break and continue).

This restriction is mainly of technical nature and probably due to the fact that the original definition dealt with recognizing DOALL loops in FORTRAN programs only.

Our goal is to define reduction operations independent of the used language and independent of the loop structure.

5.2.1.3 First Approach of Generalized Reduction Recognition

In the setting of $ParA_\gamma$ it is not possible to rely on program syntax as $ParA_\gamma$ is working only at the *LLVM* Bitcode level. Furthermore, we do not want to limit ourselves artificially to reducing the dimensionality of a vector or array.

Our definition of reduction, Red_{gen}, is a generalization of the definitions of Red_{dep} and Red_{ERS}: consider the dependence graph depicted in Figure 5.8. It shows the basic dependence pattern that appears in a reduction operation. Dependences δ^{lc}_{flow} and δ^{lc}_{out}

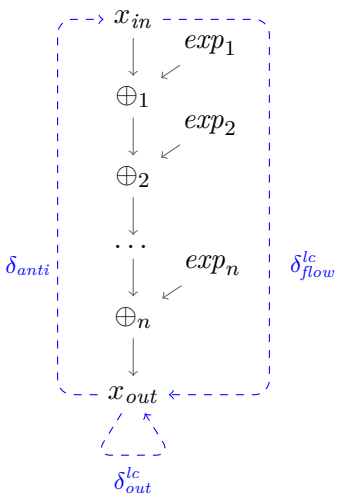


FIGURE 5.9: General form of a *redChain*. Solid arrows depict data flow, dashed arrows reduction relevant data dependences.

are loop carried and can basically be ignored upon successful recognition of the reduction, provided corresponding fixup code is added, e.g., after the loop.

Combining the dependence based approach with the concept of an ERS, we get a general notion of a reduction chain (*redChain*) as depicted in Figure 5.9.

Intuitively a value x is a reduction value in region R if it enters the region via an input node of x (x_{in} in the example). Examples for input nodes are load instructions or loop-carried ϕ nodes, in case R represents a loop. The value is combined with arbitrarily many expressions ($exp_1 \ldots exp_n$) using the same number of operators ($\oplus_1 \cdots \oplus_n$). It leaves the dynamic instance of R through a corresponding output node (x_{out}), for example a store, a loop carried ϕ node or any node used outside of R. A chain can split up and end in multiple output nodes; this for example happens naturally in loops with *continue* statements between different accesses to the reduction value. Furthermore, a chain can branch and join in ϕ nodes, provided x flows into each of the operators exactly once. No intermediate value of a chain is allowed to be used outside of R or outside of a valid *reduction chain* within R. Multiple independent chains on the same reduction value can exist in the same region.

Only if these conditions are fulfilled, we can safely "ignore" the loop carried dependences δ^{lc}_{flow} and δ^{lc}_{out}, as mentioned earlier, provided we add fixup code if at least one of them

might be violated by parallel execution.

Note that, since $ParA_\gamma$ is working on an SSA based intermediate representation, this definition already includes the concept of following the def-use chains covered by ERSs as local variables do not appear as loads and stores to different addresses in the chain. Instead, each intermediate result on the chain may or may not have been assigned to a local variable at the source level. Both source representations, one assigning intermediate values to local variables and one not doing so, are normalized to the same form in LLVM's IR.

5.2.1.4 Notation

Before we get to the final definitions, we need to introduce some further notation.

The following definitions will be based on SSA based control flow graphs (*CFGs*). The nodes N of such a graph G represent instructions (or operations); a *CFG* further contains two different sets of edges:

$E : N \times N$ is the set of control flow edges, denoted by $a \to b$ with the meaning that there exists a path in G on which an operation $b \in N$ is executed immediately after an operation $a \in N$.

$\mathring{E} : N \times N$ is the set of data-flow (or def-use) edges, denoted by $s \mathring{\to} t$ meaning that data produced by an operation $s \in N$ is directly consumed by an operation $t \in N$.

$a \to^\star z$ ($a \mathring{\to}^\star z$) denotes the set of all control-flow (def-use) paths from an operation a to another operation z. It is a subset of the transitive closure of E ($\mathring{E}$).

By $\oplus \in p$ for $p \in s \mathring{\to}^\star t$ we state that $\oplus$ is an operation on a def-use path p starting from s and ending in t.

Further, $a \to^\star_R z$ ($a \mathring{\to}^\star_R z$) denotes the set of paths not leaving a region R, which in turn is defined in terms of its contained nodes ($R \subseteq N$):

$$a \to^\star_R z := \{p \in a \to^\star z | \nexists n \in p.n \notin R\}$$

and, respectively

$$a \overset{\circ}{\rightarrow}^{\star}_{R} z := \{p \in a \overset{\circ}{\rightarrow}^{\star} z | \nexists n \in p.n \notin R\}$$

If A is a set of nodes, then $A \rightarrow^{\star}_{R} z$ ($A \overset{\circ}{\rightarrow}^{\star}_{R} z$) is the set of all paths from any node in A to z:

$$A \rightarrow^{\star}_{R} z := \bigcup_{a \in A} a \rightarrow^{\star}_{R} z$$

$\sigma(p)$ is the source of a path p and $\tau(p)$ its target. $ops(\oplus)$ denotes the set of operands of $\oplus$ and $uses(\oplus)$ the users of the value computed by $\oplus$:

$$\forall p \in a \rightarrow^{\star} z.\sigma(p) := a$$

$$\forall p \in a \rightarrow^{\star} z.\tau(p) := z$$

$$ops(\oplus) := \{n \mid n \overset{\circ}{\rightarrow} \oplus \in \mathring{E}\}$$

$$uses(\oplus) := \{n \mid \oplus \overset{\circ}{\rightarrow} n \in \mathring{E}\}$$

5.2.1.5 Definition

In the following we formally define the notion of a *redChain*.

Definition 5.5 (*RedOps*). *RedOps* denotes the set of all reduction operators; a reduction operator $\oplus$ is any associative and commutative operation.

In principle, associativity is enough to relax the involved dependences to allow for limited parallel execution. In order to be able to "remove" the dependences, as motivated before, we need commutativity as well.

In our setting, we implemented the recognition and realization of the following reduction operators: +, *****, **min**, **max**, **and**, **nand**, **or** and **xor**. Note that although **-** and **/** are not associative and commutative, they can still be accepted by moving them from the

reduction term to the *exp* term. A + or * operation is virtually introduced in their place respectively.

Definition 5.6 (Compatible *redOp*). Two reduction operators (including the *pseudo*-RedOps - *and /*) are compatible to each other if they can be replaced by each other in the top-level reduction expression. The set of reduction operators compatible to a given operator $\oplus$ is denoted as $\Gamma_\oplus$.

Furthermore, we define the set of input nodes, i.e., the nodes via which the reduced value (x) can flow into a *redChain* in a code region R:

Definition 5.7 ($in_R(x)$).

$$in_R(x) := load_R(x) \cup \Phi_R^{lc}(x) \cup \{x\}$$

$in_R(x)$ contains all loads of the value x that appear in region R; if R represents a loop, then $in_R(x)$ also contains the nodes in $\Phi_R^{lc}(x)$, the set of ϕ-nodes carrying x along the loop; finally, it contains x itself, regardless of the fact if x is part of R or not. This last part is crucial if R does not represent a loop and there is no ϕ-node and not necessarily a load involved.

Similarly define $out_R(x)$ as the set of nodes via which the reduced variable x can leave an instance of the region R:

Definition 5.8 ($out_R(x)$).

$$out_R(x) := store_R(x) \cup lcPhiOps_R(x) \cup extUsed_R(x)$$

where

$$lcPhiOps_R(x) := \{o \mid \phi \in \Phi_R^{lc}(x) \wedge o \in ops(\phi) \wedge o \in R\}$$

$$extUsed_R(x) := \{o \mid in_R(x) \overset{\circ}{\rightarrow}{}_R^\star o \neq \emptyset \wedge uses(o) \setminus R \neq \emptyset\}$$

$onChain_R(x)$ denotes the set of nodes (operations) being part of at least one def-use path of x in R:

Definition 5.9 ($onChain_R(x)$).

$$onChain_R(x) := \{o \mid \exists p \in in_R(x) \overset{\circ}{\rightarrow}^{\star}_R out_R(x).o \in p\}$$

Finally, $legalInflow_R(\otimes, x)$ is a predicate stating that a value of x enters $\otimes$ via exactly one operand, or, in case of a ϕ operator, via *all* operands.

Definition 5.10 ($legalInflow_R(\otimes, x)$).

$$legalInflow_R(\otimes, x) := \left(\mid ops(\otimes) \cap onChain_R(x) \mid = \begin{cases} \mid ops(\otimes) \mid, & \text{if } \otimes = \phi \\ 1, & \text{otherwise} \end{cases} \right)$$

We now define the predicate $Red_{gen}(x, \oplus, R)$ meaning that code region R contains a reduction over variable x using only reduction operation $\oplus$, or compatible reduction operations:

Definition 5.11 (Red_{gen}). For $Red_{gen}(x, \oplus, R)$ to be a valid reduction, all of the following conditions have to be fulfilled:

1. $\oplus \in RedOps$
2. $\forall s \in out_R(x).in_R(x) \overset{\circ}{\rightarrow}^{\star}_R s \neq \emptyset$
3. $\forall p \in in_R(x) \overset{\circ}{\rightarrow}^{\star}_R out_R(x)\ .\ \forall p' \in \sigma(p) \rightarrow^{\star}_R \tau(p).\nexists s \in \left(p' \cap out'_R(x)\right).s \neq \tau(p)$
   $\wedge \forall \otimes \in p\ .\ \otimes \in \left(\Gamma_{\oplus} \cup \{\phi\}\right)$
   $\wedge\ legalInflow_R(\otimes, x)$

where

$$out'_R(x) := out_R(x) \setminus lcPhiOps_R(x)$$

The first condition ensures that the used operator $\oplus$ is indeed a reduction operator.

The second condition ensures that any observable value stored in the same location as x is computed from an earlier version of x. This basically forbids simple assignments of the form $x := const$ which would not be commutative.

The third condition ensures that no intermediate values are allowed to escape the region R, that all *RedOps* on x in R are compatible, and that multiple values of x are allowed to flow into a path only via ϕ-nodes.

Please note that while most definitions so far are based on def-use paths, this very last condition is based on control-flow paths between the source and sink of a *reduction chain*. This is crucial, as any out value of x, not necessarily being part of the same reduction chain, that is computed on any path between the source and sink of a realized *reduction chain* qualifies as the exposure of an intermediate value of x. Also, note that this condition effectively prohibits interleaved reduction chains on the same reduction location.

In the last condition (3), $out'_R(x)$ had to be used instead of $out_R(x)$ for quite a subtle reason. Consider the code in Listing 5.7: due to the conditional *continue* statement, there exists a *redChain* containing multiple nodes observable from outside of one dynamic instance of the chain. The chain containing all updates to x also contains an operand to the loop carrying ϕ, which is part of $out_R(x)$; but $out_R(x)$ also contains the last update operation to x on the very same chain.

As we know, by the nature of the loop, that only the last instance of a value that is only carried by the ϕ will ever be observable from the outside of the loop, we allow more than one observable node in that case, provided that $\tau(p)$ is the only one not being loop-carried.

The set of reduction chains $redChains(x, \oplus, R)$ is then defined as follows:

Definition 5.12 (*redChains*).

$$redChains(x, \oplus, R) := \begin{cases} in_R(x) \overset{\circ}{\rightarrow}{}^{\star}_R\ out_R(x), & \text{if } Red_{gen}(x, \oplus, R). \\ \emptyset, & \text{otherwise.} \end{cases}$$

This completes our basic definition Red_{gen}.

The presented definition generalizes the definitions mentioned in the earlier sections: first of all, it is not limited to a particular syntactical appearance of the code. Second, it allows several expressions to flow into the reduction chain; moreover, it allows the reduction code to be spread all over its region R. Also, note that this definition allows the *redChains* to

split up (i.e., one load flows into multiple stores), or being joined (multiple paths being joined by control-flow via ϕ-nodes), or to spread over multiple disjoint chains.

The region R is crucial to the definition of a reduction. In the literature, R usually represents a loop. In particular, almost in all cases a loop, that is candidate for *DOALL* style parallel execution, once the reduction has been verified. The definition of Red_{gen} in turn does not require R to represent a loop, although this will be the case in almost all interesting situations. Nevertheless, reduction recognition in non-loop code can lead to more parallelization opportunities as well and is covered by the definition for the sake of completeness.

More details on the consequences of a successfully recognized reduction will be discussed in Section 5.2.1.7.

5.2.1.6 Variance of a reduction

One important property of a reduction when it comes to profitable realization of the reduction is the so-called *variance*. A *reduction chain* is called *varying* in its containing region R if the reduction location is not a simple variable being understood as an identifier naming a fixed memory cell, but instead can refer to different locations during the course of execution of one dynamic instance of R. The range in which the reduction location potentially varies is called the chain's *variance*.

Consider the *BiCG* implementation in Figure 1.4a. The reduction in the $s[j] = s[j] + \ldots$ statement is represented by a varying chain in its containing i-loop as this statement accesses different locations ($s[0] \ldots s[NY - 1]$) in every iteration of the i-loop. The $q[i] = q[i] + \ldots$ statement in contrast is represented by a non-varying chain in the j-loop.

5.2.1.7 Consequences of Reductions

Once all *reduction chains* for a reduction location x have been identified for a region R, all dependences between different dynamic instances of these *reduction chains* can be relaxed,

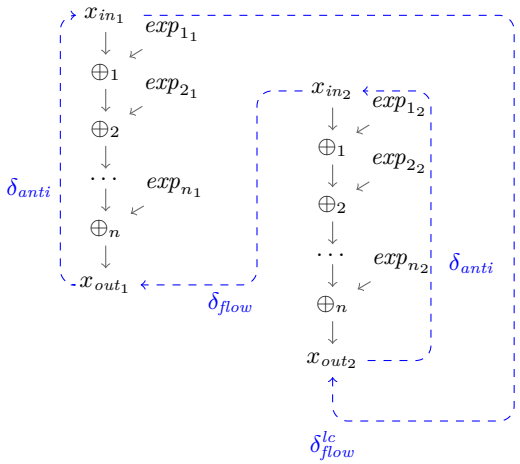


FIGURE 5.10: Multiple chains in $redChains(x, \oplus, R)$. Solid arrows depict data flow, dashed arrows reduction relevant data dependences.

provided corresponding fix-up code is added before and after the region R. To further generalize the picture of Figure 5.9, take a look at Figure 5.10.

The figure contains two reduction chains for region R; the right chain appears after the left one in R and relies on the result computed by the left chain. There is a complete chain of dependences from x_{in_1} to x_{out_2}, effectively linearizing the whole computation. Further, assuming R represents a loop, the picture shows a loop-closing flow dependence from x_{in_1} to x_{out_2}. In case R does not represent a loop, this last dependence would simply not exist.

The successful recognition of all *reduction chains* to x allows to relax certain involved dependences. A special case of such relaxation is the removal of a loop-carried flow-dependence if R corresponds to a loop. The accesses to x in R which induced the corresponding dependence are marked in the PDG and the scheduler is allowed to break those dependences. If it does so, we call the involved *reduction chains realized* and introduce fix-up code for those chains during final code generation, i.e., at runtime.

5.2.1.8 Fixup Code

For every realized reduction chain fixup code needs to be added to ensure that code outside of the region R sees the correct accumulated values. Several approaches exist in the literature to perform a successful reduction. Which method is applicable depends

mostly on the characteristics of the region R; which ones are profitable depends on several criteria, in particular on the access patterns to the reduction variable x.

Yu et al. [36] give a short overview of existing reduction algorithms and reference the corresponding work. We refer the interested reader to their work to find out more about different algorithms, a qualitative comparison thereof and a dynamic selection scheme.

In this work, we employ the following, universally applicable scheme. For every realized reduction chain, we modify the code using one of two approaches: privatization based, or atomic section based. The selection of either scheme depends on the variance of the reduction location.

To fix non-varying realized *reduction chains* as well as varying ones whose variance is fixed prior to entering the region and can thus be evaluated before entering, $ParA_\gamma$ uses privatization:

1. Right before the parallel section containing the reduction region, code is inserted to allocate a private copy of the reduction location for each thread of parallel execution. The necessary size of one private copy depends on the variance and might well require to privatize whole arrays or complex data structures.

   The (maximum) number of private copies depends on the used parallelization scheme but is typically conservatively bound to the size of the thread pool of the dynamic scheduler executing the parallel tasks.

   The private copies are allocated in cache line[5] aligned locations in order to prevent interference of the individual threads acting on their respective private copies. This optimization is crucial to get decent performance.

2. Next, code is placed to initialize all private copies to the neutral element of the respective reduction operation. Depending on the type of the reduction location and the neutral element this is either a single *memset* of the whole region to 0, or a loop individually initializing each private copy.

[5] A typical cache line size is 64 byte.

3. Code performing a load from the respective private reduction location, followed by the reduction operation and a store to the private copy is inserted immediately after every node in $out_R(x)$, reducing its result (store instructions are replaced).

   Note that in the course of runtime code generation and optimization, the individual loads and stores from and to the private reduction locations are removed and joined to one load and store respectively if the thread executes multiple reduction operations consecutively. This in particular holds true for the dynamic blocking of parallelized loops as described in Section 7.3.

4. Finally fix-up code is placed right after the reduction region. Privatized copies of the reduction location are joined into one reduction value again. Furthermore, if necessary, the private copies of the reduction location are freed again.

5. Every user u of a replaced or extended reduction value is rewired to use the constant neutral reduction element instead (if $u \in R$), or load the reduced result from the reduction address (if $u \notin R$).

For varying chains with an unknown variance, $ParA_\gamma$ uses atomic operations without the need for privatization but at the cost of the atomic operations.

1. Every node $in_R(x)$ is replaced by the neutral element corresponding to the reduction operation.

2. Code performing an atomic fetch and reduce operation is inserted immediately after every node in $out_R(x)$, reducing its result (store instructions are replaced). Depending on the reduction operator and the execution platform this might be a single instruction (e.g., atomic compare and add), or a region of code involving a compare and set instruction. In any case note that not the whole chain needs to be encapsulated in the atomic section. Every user u of a replaced or extended instruction is caused to use the constant neutral reduction element instead (if $u \in R$), or load the reduced result from the reduction address (if $u \notin R$).

3. Also, for every instruction in $lcPhiOps_R(x)$ a corresponding atomic fetch and reduce operation is inserted on the loop back edges corresponding to that instruction.

This last scheme particularly also works if the number of iterations of the loop and even the number of participating threads is unknown. Such a situation for example occurs in our implementation if the code to be parallelized arises from an irregular application, and parallelization is performed via a flexible work stealing task queue mechanism as provided by Intel TBB [40].

Although the scheme based on atomic operations is quite expensive and therefore mostly considered unprofitable in classic work, it is quite appealing in our setting due to its universal applicability. Apart from that, parallelizing the reduction as such is not our primary goal; instead we aim for executing the instances of the expressions being reduced in parallel. Assuming that the overhead introduced by the code for the *atomic compare and reduce* is less than the execution time of the reduced expressions, this scheme is still profitable.

Evaluating the implementation of a dynamic selection of the reduction realization scheme as motivated by Yu et Al. [36] is left for future work. In our experience however, the expected benefit of such a scheme does not seem worth the overhead in domains in which the applications typically do not spend a significant fraction of the execution time performing the reduction computation as such. In the applications we aimed at so far, the reduction usually plays a minor role combining the results of independent but long-running computations.

5.2.2 Privatization

Another very important parallelization enabling technique is privatization [79, 109] in its own terms: within a PDG subgraph R rooted in node n, a memory location x is *privatizable* if and only if the following conditions are met:

- On every control-flow path entering R and ending in a possible read access $l \in R$, there is at least one definitive write access $s \in R$ to x, and
- on every control-flow path from any possible write access $s \in R$ to any possible read $l' \notin R$, there is another definitive write to x on that path.

Note how a "rooted PDG subgraph" is equivalent to a single entry, single exit region in the control flow graph. This is an important property for efficient code generation. It allows to place potentially necessary code allocating private copies at the entrance of the region, and cleanup code at its exit.

Our definition of privatizability is neither surprising nor new. But, as with reductions, we do not resolve dependences induced by privatizable accesses prior to scheduling parallel code. Instead we annotate PDG nodes n that represent a reachable subgraph R with memory locations x to which all accesses in R are privatizable. Dependences that enter or leave a dynamic instance of R and are induced by accesses to x are then allowed to be broken by the scheduler. We call a privation opportunity on a location x in region R realized, if the scheduler indeed decides to break such a dependence.

The final code generation for a realized privatization opportunity on x in R is straight forward:

1. Right before R, code is placed to allocate a cache line aligned private copy of x for R. Note that it is not necessary to initialize the private copy to any value as the legality conditions for privatizability guarantee that it is overwritten before it may be read.
2. Accesses to x in R are rewired to use the private copy allocated in the previous step.
3. At the exit of R, the private copy is freed again, in case this is necessary.

5.2.3 Speculation

As explained earlier in Chapter 4, *Sambamba* provides two different speculation systems. During the course of this work, none of both systems has reached a state in which it has been able to dynamically provide specific information on misspeculation rates, reasons and sources. Therefore, the best we can currently do is to penalize the violation of a dependence proportionally to the probability that it will manifest at runtime: a dependence edge that the scheduler is allowed to speculatively ignore has a source and a target potentially accessing the same memory location. The speculation is guaranteed to be successful if at runtime either the source or the target statement is not executed at all. The static

scheduler accounts for speculation overhead for all dependences marked as speculatively ignorable, that the scheduler decides to break. It amounts to a value that grows linearly in the size of the commonly accessed values multiplied by the relative execution frequency of the source and target statements respectively.

In our evaluation in Chapter 9 we do not rely on speculation.

This chapter laid the groundwork for our generalized task parallelization scheme. In particular, the used program representation based on the program dependence graph has been introduced. The *PDG* completely abstracts the strict program execution order defined by the control flow graph, leaving only control and data dependences, which suffice to preserve semantics.
Furthermore, important enabling techniques have been introduced: speculation, privatization and reduction recognition form the necessary basis of automatic parallelization. The given definition of generalized reduction is agnostic with regard to a specific syntactic form of the reduction or program structure surrounding it and is solely based on the flow of the reduction value through an associative and possibly commutative reduction operation.

Chapter 6

ILP-based PDG-Scheduling

In this chapter, we describe in detail how we translate the problem of finding a schedule for parallel task execution to the optimization of an integer linear program. Due to its expressive power, the available toolset (algorithmical and technical), and its extensibility using a clean and well understood mathematical language we chose to use ILP optimization at the heart of our generalized task parallelization.

We formulate an integer linear program to compute a *local* schedule for each individual group node in the PDG. The intuition behind the following ILP formulation is to map the children $I_g = \{g[i] \mid i \in \{1..n_g\}\}$ of a group node g onto a two-dimensional grid. One dimension in this grid corresponds to time and is subdivided into *stages* (S_g) of execution; the other axis corresponds to placement and is subdivided into *threads* (T_g). $|T_g|$, the maximum number of parallel threads that the generated schedule will use, is a constant parameter to the ILP scheduler. Nodes will possibly execute in parallel at runtime if and only if they are placed in different threads of the same stage. The optimization goal of each ILP is to minimize the latency for its corresponding PDG group node.

Figure 6.1 shows a possible result of solving the ILP corresponding to the PDG group node labeled *true* in Figure 5.2. Stage 2 is a parallel stage in this schedule, spawning off the recursive call to *seqquick* while proceeding with the next iteration of the loop, which is represented by its PDG node labeled with the while statement in thread 1 of stage 2. A schedule representing a parallel loop is called a reentrant schedule. This example also

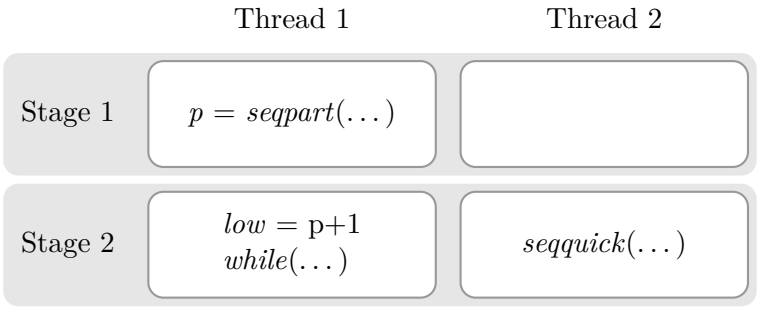


FIGURE 6.1: Possible schedule for the group node labeled *true* of the PDG in Figure 5.2. This schedule represents partial parallel execution of the loop in *seqquick* of Figure 1.4a.

shows how the scheduler transparently parallelizes loops without any special treatment[1]. A schedule for a *DOALL*-loop, for instance, would in its simplest form contain one stage with two threads of which one contains the loop body and the other the computation of the induction variable (if present).

As described in Section 5.1 dependences between subgraphs can potentially be relaxed by making use of privatization, reduction or speculation opportunities. Such relaxation allows the scheduler to execute the source and target of a dependence in parallel to each other, provided, corresponding fix-up code is added. The potential overhead introduced by such countermeasures is encoded in the ILP cost-function as described in Subsection 6.1.2.

6.1 ILP Formulation

6.1.1 Prerequisites

In favor of a shorter notation, we assume $g \in G$, $r \in R$, $d \in D$, $i, j \in \{1 \dots |I_g|\}$, $s \in \{1 \dots |S_g|\}$, and $t \in \{1 \dots |T_g|\}$ in the following discussion. Before constructing an ILP for each group g of a PDG, we compute estimates of accumulated execution costs for each PDG node as follows:

Regular Nodes. The size $\|r\|$ of a regular node r is computed by traversing the contained instructions (remember that a regular node in our case represents a basic block),

[1] As shown in Section 7.3, the code generation of $ParA_\gamma$ has special treatment of loops to generate efficient parallel code.

and accumulating their individual cost. For this purpose, we statically estimate the cost of arithmetic instructions; the cost of memory instructions is taken into account by assessing the size of written, read or copied data if statically possible. Costs of call instructions (call sites) are taken from dynamic call site profiles of earlier runs of the application (see Subsection 7.1.1), if such profiles are available, or estimated: if the called function is statically known, the call graph is traversed and the cost of transitively called functions accumulated. If the call is indirect, i.e., the called function is not statically known, we assume high cost for the call. This is in favor of parallelization and leaves it to the runtime code generation to use call site execution time profiling to find out if the assumption was beneficial or not.

Decision Nodes. $\|d\|$, the execution cost of a decision node d, is defined as the frequency-weighted sum of the accumulated size of its children:

$$\|d\| := \sum_i (\|d[i]\| * \mathit{freq}(d_i)) + |d|$$

where $\mathit{freq}(d_i)$ is the frequency of d selecting child node $d[i]$ for execution, and $|d|$ is the non-accumulated size of d (which is computed in a similar way to the cost of regular nodes).

Frequencies are statically estimated or read from profiling information persisted during earlier runs of the application. $ParA_\gamma$'s runtime system is able to collect such branch profiles (see Section 7.1).

Group Nodes. $\|g\|$, the size of a group node g, is defined as the sum of accumulated sizes of its children:

$$\|g\| := \sum_i \|g[i]\|$$

6.1.2 Constraints

For each group node g of the PDG, a directed acyclic graph $DAG_g := (I_g, \Delta_g)$ describes the data dependences and possible conflicts between the children of g. Its nodes are the

TABLE 6.1: Variables used in the ILP for a group node g.

$\forall s$:		
$\gamma_g[s] \in \mathbb{N}$	:=	critical path length of stage s
$\varphi_g[s] \in \mathbb{B}$	:=	1 if, and only if stage s is nonempty
$\forall s\, \forall t$:		
$\sigma_g[s,t] \in \mathbb{N}$	:=	size of thread t in stage s
$\varphi_g[s,t] \in \mathbb{B}$	:=	1 if, and only if thread t in stage s is filled
$\forall i\, \forall s\, \forall t$:		
$\sigma_g[i,s,t] \in \mathbb{N}$	:=	size of $g[i]$ in stage s, thread t
$\chi_g[i,s,t] \in \mathbb{B}$	:=	1 if, and only if $g[i]$ is placed in stage s, thread t
$\forall (i \to j)_g \in \Upsilon_g \cup \Omega_g \cup \Psi_g$:		
$par_g[i,j] \in \mathbb{B}$	:=	1 if, and only if $g[i]$ and $g[j]$ execute in parallel

children of g (i.e., I_g); Δ_g is the set of edges.

An edge $(i \to j)_g \in \Delta_g$ has the meaning of its source $g[i]$ depending on or conflicting with its target $g[j]$ in g. It has associated communication cost $\|(i \to j)_g\|$, being an estimate of data to be communicated if $g[i]$ and $g[j]$ execute in different stages. An edge is called fulfilled by a schedule if according to the schedule its target is executed before its source.

$\Upsilon_g \subseteq \Delta_g$ is the set of edges requiring speculation support in the source and target thread of the dependence if the scheduler breaks it; $\Omega_g \subseteq \Delta_g$ and $\Psi_g \subseteq \Delta_g$ are the reduction and privatization ignorable edges respectively. Note that Υ_g, Ω_g and Ψ_g are not necessarily disjoint. One edge might require multiple fix-up mechanisms in order to be breakable.

To reduce the number of constraints, we precompute the set of transitive edges $\Theta_g \subseteq \Delta_g$, thus $(I_g, \Delta_g \setminus \Theta_g)$ is the transitive reduction of DAG_g. We call $\widetilde{\Delta_g} := \Delta_g \setminus (\Upsilon_g \cup \Omega_g \cup \Psi_g)$ the set of *unbreakable dependences*.

Table 6.1 introduces the variables used in the ILP formulation. Note that the number of variables is quadratic in I_g since $|S_g| \leq |I_g|$ and $|T_g|$ is a constant.

Minimize

$$\sum_s \left(\gamma_g[s] + \varphi_g[s] * SInitOvhd + \sum_t \left(\varphi_g[s,t] * TInitOvhd \right) \right) + ComCost_g + RelaxP_g$$

where

$$ComCost_g := \sum_{(i \to j)_g \in \Delta_g} \sum_s \sum_t \left(max\left(\chi_g[i,s,t] - \chi_g[j,s,t], 0 \right) * \|(i \to j)_g\| \right)$$

$$RelaxP_g := \sum_{(i \to j)_g \in \Upsilon_g \cup \Omega_g \cup \Psi_g} \left(par_g[i,j] * \left(SpecP_g[i,j] + RedP_g[i,j] + PrivP_g[i,j] \right) \right)$$

SInitOvhd and *TInitOvhd* are ILP constant estimates of stage and thread initialization overhead respectively.

subject to the following constraints

Constr. 1 (Dependence Order 1)

$\forall (i \to j)_g \in \widetilde{\Delta_g} \setminus \Theta_g \colon\ SD_{i,j} * |T_g| - TD_{i,j} \geq 0$

Constr. 2 (Dependence Order 2)

$\forall (i \to j)_g \in \widetilde{\Delta_g} \setminus \Theta_g \colon\ SD_{i,j} * |T_g| + TD_{i,j} \geq 0$

Constr. 3 (Size Placement Connection)

$\forall i \, \forall s \, \forall t \colon\ \sigma_g[i,s,t] = \chi_g[i,s,t] * \|g[i]\|$

Constr. 4 (Parallel)

$\forall (i \to j)_g \in \Delta_g \colon\ par_g[i,j] * |T_g| \geq abs(TD_{i,j}) - |T_g| * abs(SD_{i,j})$

Constr. 5 (Thread Filling)

$\forall s \, \forall t \colon\ \sum_i \chi_g[i,s,t] \leq \varphi_g[s,t] * |I_g|$

Constr. 6 (Unique Placement)

$\forall i \colon\ \sum_s \sum_t \chi_g[i,s,t] = 1$

Constr. 7 (Stage Filling)

$\forall s \colon\ \sum_t \varphi_g[s,t] \leq \varphi_g[s] * |T_g|$

Constr. 8 (Thread Size)

$\forall s \, \forall t \colon\ \sigma_g[s,t] = \sum_i \sigma_g[i,s,t]$

Constr. 9 (Critical Path)

$\forall s \, \forall t \colon\ \gamma_g[s] \geq \sigma_g[s,t]$

Constr. 10 (Speculation Order)

$\forall (i \to j)_g \in \Upsilon_g \colon\ STD_{i,j} \geq 0$

---

where

$$SD_{i,j} := \sum_s \sum_t \left(s * \left(\chi_g[i,s,t] - \chi_g[j,s,t] \right) \right)$$

$$TD_{i,j} := \sum_s \sum_t \left(t * \left(\chi_g[i,s,t] - \chi_g[j,s,t] \right) \right)$$

$$STD_{i,j} := \sum_s \sum_t \left(\left(s * |T_g| + t \right) * \left(\chi_g[i,s,t] - \chi_g[j,s,t] \right) \right)$$

FIGURE 6.2: Objective function and constraints used in the ILP formulation for group node g.

Figure 6.2 shows the final ILP formulation. The used objective function minimizes the critical path execution time ($\sum_s \gamma_g[s]$) while penalizing the use of multiple stages and threads, as well as inter-thread communication. Additionally, broken dependences marked as resolvable by privatization, reduction and speculation are punished. Each used stage is modeled to introduce overhead (*SInitOvhd*), which is motivated by the setup cost of synchronization mechanisms at runtime. Each used thread causes runtime overhead (*TInitOvhd*) due to the cost of setting up and spawning parallel tasks.

ComCost accounts for introduced inter-thread and inter-stage communication by adding the statically estimated communication cost $\|(i \to j)_g\|$ per boundary crossing dependence. This estimate is solely based on the communication volume (i.e., the static size of the communicated data) in the current implementation. A dependence $(i \to j)_g \in \Delta_g$ is considered boundary crossing if, and only if, $\chi_g[i, s, t]$ and $\chi_g[j, s, t]$ differ for any given pair of s and t in the current solution of the ILP.

The speculation penalty cannot model the actual overhead of speculation statically: the main cause for overhead are frequent rollbacks and re-executions, which are inherently input dependent. Instead, the speculation penalty is approximated by an ILP-constant overhead $SpecP_g[i, j]$ per speculatively ignored dependence crossing a thread boundary. It is computed based on the execution frequencies of the PDG nodes causing the conflict represented by the dependence. *RelaxP* accounts for this by adding $SpecP_g[i, j]$ for each speculatively ignored edge $(i \to j)_g \in \Upsilon_g$.

Reduction $RedP_g[i, j]$ and privatization $PrivP_g[i, j]$ penalties are treated similarly to the speculation penalties. But the individual penalty per broken dependence differs: in case of privatization, the penalty grows linearly with the size of the value to be privatized. Similarly, the reduction penalty grows linearly with the size of the reduction variable. In case of a varying chain, this size is multiplied by the chain's variance.

The constraints of the ILP can be partitioned into two groups: The first group is formed by the Constraints 6, 1, 2 and 10, which are used to model legality constraints to preserve the semantics of the program:

- Constraint 6 ensures that each node is scheduled exactly once. It guarantees *unique placement* of nodes.

- Constraints 1 and 2 ensure for each unbreakable dependence $(i \to j)_g$ which is not marked transitive, that execution of $g[j]$ precedes execution of $g[i]$. This means that $g[j]$ is executed in an earlier stage than $g[i]$, or in the same stage and thread. Note that $SD_{i,j}$ denotes the stage-distance between the placement of $g[i]$ and $g[j]$. The constraints ensure that $SD_{i,j}$ is non-negative, and if it is zero, then also $TD_{i,j}$ (the thread-distance) must be zero.

- Constraint 10 ensures that for each speculatively ignorable dependence $(i \to j)_g$, $g[j]$ is executed in an earlier stage than $g[i]$, or in the same stage, but possibly different thread with a lower number. The fact that there still are restrictions on speculatively ignored dependences is due to the conflict detection and recovery mechanism of the runtime system. It allows for conflict detection only between different threads of the same stage. Thus, $g[j]$ is not allowed to be placed in a later stage than $g[i]$. The requirement that $g[j]$ needs to be in a thread with a lower number t than $g[i]$ is necessary to guarantee a non-cyclic commit order between the threads.

The second group is formed by the Constraints 4–9, which model execution cost and are used in the objective function:

- Constraint 4 defines the $par_g[i,j]$ variables which are used in the cost function to penalize dependences broken using speculation, privatization or reduction.

- Constraints 5 and 7 define the $\varphi_g[s,t]$ and $\varphi_g[s]$ variables to reflect if a thread or stage is filled, i.e., contains at least one node.

- Constraint 8 models the size $\sigma_g[s,t]$ of a thread as the sum of sizes of contained nodes.

- Constraint 9 models the critical path length $\gamma_g[s]$ of a stage as the size of the largest contained thread: it is pulled down as it is used in the objective function, but guaranteed to stay larger than the size of any contained thread.

The remaining Constraint 3 connects both groups of constraints by relating the boolean variables $\chi_g[i, s, t]$ to the corresponding size variables $\sigma_g[i, s, t]$. Further it defines the size of node $g[i]$ as being the ILP constant $\|g[i]\|$.

Note that no further constraints are required for reduction and privatization resolvable edges. A dependence which is resolvable by privatization or reduction, can, in terms of legality, simply be ignored. It only needs to be reflected correspondingly in the cost of the resulting schedule. This is taken account for in the *RelaxP* definition of the cost function. The number of constraints is quadratic in the number of child nodes per group.

Further, remember that Constraint 6 guarantees that each node is uniquely placed, i.e., not duplicated among parallel threads. As shown in detail in Subsection 6.2.2, it is possible to relax this restriction for nodes which are safe to clone: Every node that is guaranteed not to execute any observable side-effect (program termination, system calls or memory writes) can be duplicated. However, such relaxation severely increases the complexity of other constraints whose formulation relies on every node to be placed exactly once.

The ILP solver can be initialized with the sequential schedule: $\chi_g[i, s, t]$ is set to 1 for $s = 0$ and $t = 0$, 0 otherwise. Providing an initial solution effectively speeds-up the optimization process in practice.

6.2 Alternative ILP Formulations

When thinking about the ILP formulation above and the intuitive ideas described in the previous sections, alternative formulations, or extensions, of the ILP might come to mind. Two particularly interesting extensions have been implemented[2]: a *global ILP formulation* optimizing the whole function at once; and an extension allowing *code without observable side-effects to be cloned* to foster parallelization and reduce the necessary communication overhead by allowing for re-computation within a parallel task. We shortly describe the

[2]Note that the implementation of the changed ILP contained in $ParA_\gamma$ does not exactly match the formulations described in the following, which, for didactic reasons, have been chosen to be as close as possible to the ILP formulation in Subsection 6.1.2. The actual implementation which is still contained in $ParA_\gamma$ has not been actively maintained and did not undergo all changes of the final ILP formulation.

necessary extensions to the original ILP formulation and explain why both extensions have not been included in the final approach.

6.2.1 Whole Function Scheduling

The idea is straight forward: instead of determining one local schedule per group node of a PDG representing the function to parallelize, one can determine a single globally optimal schedule for the whole function by solving one, of course more complex, ILP. One possibility to extend the ILP formulation is described in the following. The computation of the accumulated execution cost of PDG group and decision nodes prior to formulating the ILP is not necessary anymore and is replaced by sub-ILPs.

6.2.1.1 Required Changes

The cost of a group node g is reflected in the new global ILP by instantiating the complete local ILP_g for g as described in the previous section. $\|g\|$ as described in Subsection 6.1.1 is superseded by the cost function of ILP_g denoted as $\|g\|_{ILP}$. Note that this notation denotes the cost function itself and not its value:

$$\|g\|_{ILP} := \sum_s \left(\gamma_g[s] + \varphi_g[s] * \mathit{SInitOvhd} + \sum_t \left(\varphi_g[s,t] * \mathit{TInitOvhd} \right) \right) + \mathit{ComCost}_g + \mathit{RelaxP}_g$$

Additionally, the accumulated size of a decision node d is replaced by a linear formula (again, not its value) computing the weighted sum of its children, all being group nodes by definition of the PDG:

$$\|d\|_{ILP} := \sum_i \left(\|d[i]\|_{ILP} * \mathit{freq}(d_i) \right) + |d|$$

Finally, the cost $\|r\|_{ILP}$ of a regular node r is equivalent to the previously described accumulated cost $\|r\|$, which in turn is equivalent to the non-accumulated cost $|r|$, as regular nodes do not have any children:

$$\|r\|_{ILP} := \|r\|$$

Finally, everything needs to be connected. The first idea would be to replace all usages of $\|d\|$ and $\|r\|$ of decision and regular nodes in Constraint 3 of the ILP instantiation ILP_g

for each group node g by $\|d\|_{ILP}$ and $\|r\|_{ILP}$ respectively. Unfortunately, the result would not be linear due to the multiplication with $\chi_g[i,s,t]$.

Instead Constraint 3 is replaced by the following two constraints:

Constr. 11 (Child Size Sum)

$$\forall i\colon \ \|\mathrm{g}[i]\|_{ILP} = \sum_s \sum_t \sigma_g[i,s,t]$$

accompanied by

Constr. 12 (Unique Placement 2)

$$\forall i\, \forall s\, \forall t\colon \ \chi_g[i,s,t] * M \geq \sigma_g[i,s,t]$$

where M is a big numerical constant, guaranteed to be bigger than all $\sigma_g[i,s,t]$. Together these constraints define exactly one $\sigma_g[i,s,t]$ to correspond to $\|g[i]\|$, namely the one for which $\chi_g[i,s,t] = 1$ holds true.

The optimization goal of the global ILP is to minimize $\|root_{PDG}\|_{ILP}$, the critical path execution time of the root node of the PDG.

6.2.1.2 Implications

Using one single ILP per function seems appealing and indeed has been our initial attempt. It is not used in the final implementation of $ParA_\gamma$ as the quality of the resulting parallelism has been behind that of the local versions with the flexibility they provide at runtime: the local versions naturally expose more parallelism, and thus more possibilities to choose from at runtime, as the global version has no reason to expose parallelism in regions/groups not contributing to the critical path of execution. If the statically determined global schedule was based on imprecise performance predictions or profiles, however, static assumptions on which code is on the critical path might deem wrong at runtime.

6.2.2 Scheduling with Code Duplication

Equally appealing and indeed a frequently used technique in parallelization is the duplication of side-effect free code to trade (parallel) re-computation for linear computation and communication of the computed values. This is reflected as an extension to the ILP formulation as follows.

6.2.2.1 Required Changes

A child $g[i]$ (basic block or a whole reachable sub-graph) of group node g is considered safe to clone ($stc(g[i])$) if, and only if

- it does not write to memory (it may read memory though),
- it does not have any non-memory side-effect,
- it does not possibly terminate the containing function, and
- it is not reentrant within its containing control-dependence group g.

To understand the requirement of non-reentrancy, remember Figure 5.5 showing a node which is reentrant within its containing group g. Duplicating this node (i.e., the sub-graph reachable from it, which contains g itself) produces a program containing a loop which, on every iteration, spawns two instances of itself executing all remaining iterations. This exponentially exploding behavior is clearly undesirable and useless.

We denote by $\Sigma_g \subseteq I_g$ the set of children of g for which $stc(g[i])$ holds.

To reflect the possibility to duplicate $g[i]$, Constraints 6, 4, 1, 2, and 10 need to be adapted as follows:

Constraint 6 (Unique Placement) is replaced by the following version:

Constr. 13 (Unique Placement)

$$\forall i \in I_g \setminus \Sigma_g : \sum_s \sum_t \chi_g[i,s,t] = 1$$

augmented by its counterpart for clonable nodes:

Constr. 14 (Guaranteed Placement)

$$\forall i \in \Sigma_g : \sum_s \sum_t \chi_g[i,s,t] \geq 1$$

Constraint 4 (Parallel) is replaced by the version for dependences from one unclonable child to another unclonable child:

Constr. 15 (Parallel Unique)

$$\forall (i \to j)_g \in \Delta_g : \boxed{\left(\neg stc_g(\mathrm{g}[i]) \wedge \neg stc_g(\mathrm{g}[j])\right) \to} \; par_g[i,j] * |T_\mathrm{g}| \geq abs(TD_{i,j}) - |T_\mathrm{g}| * abs(SD_{i,j})$$

augmented by the versions with exactly one clonable child. Note that constraint 4 is only relevant in the context of relaxable dependences involving reduction, speculation or privatization. In that context, not both, source and target of the dependence can be safe to clone, as otherwise no such dependence does exist.

Constr. 16 (Parallel Clonable Source)

$$\forall (i \to j)_g \in \Delta_g : \boxed{\left(stc_g(\mathrm{g}[i]) \wedge \neg stc_g(\mathrm{g}[j])\right) \to} \; par_g[i,j] = \sum_s \sum_t \min(\chi_g[i,s,t], \sum_{t_2 \neq t} \chi_g[j,s,t_2])$$

Constr. 17 (Parallel Clonable Destination)

$$\forall (i \to j)_g \in \Delta_g : \boxed{\left(\neg stc_g(\mathrm{g}[i]) \wedge stc_g(\mathrm{g}[j])\right) \to} \; par_g[i,j] = \sum_s \sum_t \min(\chi_g[j,s,t], \sum_{t_2 \neq t} \chi_g[i,s,t_2])$$

Note that the implication is not part of the ILP, the notation has been chosen for reasons of a clearer definition. The premise (in the dashed box) is instead checked during ILP construction and the corresponding constraint (the consequence in the implication above) generated.

This new definition of $par_g[i,j]$ also requires to change the range of $par_g[i,j]$:

$\forall (i \to j)_g \in \Upsilon_g \cup \Omega_g \cup \Psi_g$:		
$par_g[i,j] \in \mathbb{N}$	:=	the number of parallel instances of i and j

The usages of $par_g[i,j]$ in the ILP do not need to be changed. The new definition of $par_g[i,j]$ is used to introduce a relaxation penalty for every pair of threads with i and j being scheduled in parallel to each other. Counting instances of parallel execution of i and j reflects the fact that fixup code needs to be introduced for every instance.

Constraints 1 and 2 (Dependence Order 1 and 2) have to be replaced due to their usage of the definitions of $SD_{i,j}$ and $TD_{i,j}$, which assume unique placement. Constraints 1 and 2 need to be replaced by a formulation with increased complexity for dependences involving at least one clonable node:

Constr. 18 (Unique Dependence Order 1)

$$\forall (i \to j)_g \in \widetilde{\Delta_{\mathrm{g}}} \setminus \Theta_{\mathrm{g}} \colon \boxed{\left(\neg stc_g(\mathrm{g}[i]) \wedge \neg stc_g(\mathrm{g}[j])\right) \to} \; SD_{i,j} * |T_{\mathrm{g}}| - TD_{i,j} \geq 0$$

Constr. 19 (Unique Dependence Order 2)

$$\forall (i \to j)_g \in \widetilde{\Delta_{\mathrm{g}}} \setminus \Theta_{\mathrm{g}} \colon \boxed{\left(\neg stc_g(\mathrm{g}[i]) \wedge \neg stc_g(\mathrm{g}[j])\right) \to} \; SD_{i,j} * |T_{\mathrm{g}}| + TD_{i,j} \geq 0$$

Constr. 20 (Clonable Dependence Order)

$$\forall (i \to j)_g \in \widetilde{\Delta_{\mathrm{g}}} \setminus \Theta_{\mathrm{g}} \colon \boxed{\left(stc_g(\mathrm{g}[i]) \vee stc_g(\mathrm{g}[j])\right) \to} \; \forall s \, \forall t \colon \chi_g[i,s,t] \leq \chi_g[j,s,t] + \sum_{s'<s} \sum_{t'} \chi[j,s',t']$$

One important additional constraint needs to be added to guarantee correctness if the target $g[j]$ of a memory-induced dependence $(i \to j)_g$ is clonable. In that case, it might be possible that $g[i]$ writes to memory that is read by $g[j]$, which introduces a new dependence $(i \to j')_g$ from $g[i]$ to any clone $g[j']$ of $g[j]$. The following constraint guarantees that no such clone is scheduled in parallel to or after $g[i]$ (or any clone thereof):

Constr. 21 (Clone Order Soundness)

$$\forall (i \to j)_g \in \widetilde{\Delta_{\mathrm{g}}} \setminus \Theta_{\mathrm{g}} \colon \boxed{stc_g(\mathrm{g}[j]) \to} \; \forall s \, \forall t \colon (1 - \chi_g[i,s,t]) * |S_{\mathrm{g}}| * |T_{\mathrm{g}}| \geq \sum_{t_2 \neq t} \chi[j,s,t_2] + \sum_{s_2 > s} \sum_{t_3} \chi[j,s_2,t_3]$$

Constraint 10 (Speculation Order) needs to be adapted similarly due to its usage of $STD_{i,j}$, which also assumes unique placement:

Constr. 22 (Unique Speculation Order)

$$\forall (i \to j)_g \in \Upsilon_{\mathrm{g}} \colon \boxed{\left(\neg stc_g(\mathrm{g}[i]) \wedge \neg stc_g(\mathrm{g}[j])\right) \to} \; STD_{i,j} \geq 0$$

is accompanied by a dedicated version for a clonable source or target:

Constr. 23 (Clonable Speculation Order)

$$\forall (i \to j)_g \in \Upsilon_{\mathrm{g}} \colon \boxed{\left(\neg stc_g(\mathrm{g}[i]) \wedge \neg stc_g(\mathrm{g}[j])\right) \to} \; STD_{i,j} \geq 0$$

And finally, similarly to constraint 21, an additional constraint is needed to guarantee soundness in case of a clonable target of a speculative dependence:

Constr. 24 (Speculative Clone Order Soundness)

$$\forall (i \to j)_g \in \widetilde{\Delta_{\mathrm{g}}} \setminus \Theta_{\mathrm{g}} \colon \boxed{stc_g(\mathrm{g}[j]) \to} \; \forall s \, \forall t \colon (1 - \chi_g[i,s,t]) * |S_{\mathrm{g}}| * |T_{\mathrm{g}}| \geq \sum_{t_2 > t} \chi[j,s,t_2] + \sum_{s_2 > s} \sum_{t_3} \chi[j,s_2,t_3]$$

Note the subtle difference between constraints 21 and 24: while in the former, no clone $g[j']$ is allowed to be scheduled in parallel or after $g[i]$ (or a clone thereof), the latter allows to execute any $g[j']$ speculatively in parallel to $g[i]$ (or a clone thereof).

6.2.2.2 Implications

While the possibility of code duplication seems appealing, we have not encountered a case in which it was critical to enable the desired parallelism. Since the heavily increased ILP complexity and the additional freedom of the scheduler instead increased the ILP solution time, code duplication is disabled by default and its activation left as a command line switch in $ParA_{\gamma}$.

6.3 Scheduling Time

As solving integer linear programs is NP-hard in general, our scheduling approach often takes a considerable amount of computation time. Remember that the described *ILP*

TABLE 6.2: (Excerpt of Table 9.1) Complexity of the programs used to evaluate $ParA_\gamma$.

Program	$SLOC$	$N_{max/avg}$	$E_{max/avg}$
alignment	612	93/9.86	128/9.27
cilksort*	387	22/8.58	23/7.19
fft	3168	63/8.92	161/10.93
blackscholes	393	24/7.38	30/6.87
BiCG	1586	31/9.15	37/9.50
gesummv	1582	24/7.88	31/8.08

is solved for each group of equally control dependent nodes and its complexity grows quadratically in the number of nodes (or linearly in the number of dependence edges) in such a group. This consequently means that the complexity of parallelizing an application in $ParA_\gamma$ is dominated by its maximum number of equally control dependent nodes, which does—like the average size of a basic block—not necessarily correlate with the program size. The latter only linearly influences the complexity. Table 6.2, which is an excerpt of Table 9.1, provides insights into the maximum and average numbers of equally control dependent nodes ($N_{max/avg}$) as well as dependences ($E_{max/avg}$) of the programs used for our evaluation in Chapter 9.

Furthermore, although an increasing number of dependence edges does indeed increase the number of constraints, this does not mean that the solving time dramatically increases. This is due to the fact that an increasing number of dependences leaves less freedom to the scheduler. IBM Cplex, the scheduler we use, is able to dramatically reduce the size of the ILP before actually solving the corresponding LP.

In practice, we found that for the majority of instances ($> 80\,\%$) an optimal solution is found in less than 10 seconds. To counter much longer execution times, we implemented several means to limit their influence.

As can be seen in Table 6.2, the most complex of the benchmarks is *fft*. Its highest number of dependences per group is 161 dependence edges after transitive reduction (324 edges before). Running the ILP until the optimum is proved, takes 4290 seconds on our machine. However, the optimal solution is found after 150 seconds, a solution within a 10% range of the optimal solution after 20 seconds.

In all our benchmarks, the solution could not be significantly improved after three minutes, so we assume that after a timeout of three minutes the best solution found so far is close enough to the optimum and interrupt the solver. This timeout is configurable.

Additionally, the generated schedules of each function are written to disk for reuse on the same machine as described in Subsection 8.2.2. This greatly reduces compilation time during frequent recompilations on a developer machine—changed dependences (and consequently PDGs) of the application lead to rescheduling of affected functions only. Optionally, the schedules can also be cached in a shared *schedule cloud* (see Subsection 8.2.3). During idle periods the cloud server can always take partial (i.e., feasible but not optimal) solutions and improve them towards an optimal one.

Note that the techniques described in this subsection are of purely practical relevance and are not critical to the approach in general.

This chapter gave an intuition and formally translated the problem of finding candidate code regions for parallel execution to an integer linear optimization problem. The mathematically sound and clean representation allows to model multiple sources of overhead introduced by parallelization and leaves the decision on if and where to use enabling techniques and introduce the corresponding overhead to an informed scheduler giving an optimality argument with respect to the chosen cost function.
Instantiation of the statically found parallelization candidates is left to an adaptive runtime system, which decides if, how, and when to actually parallelize based on actual properties of the execution environment.

Chapter 7

Runtime-Adaptive Parallel Execution

The parallelism as exploited by $ParA_\gamma$ at compile-time has two important properties:

1. It strongly favors parallel over sequential execution: the compile-time cost-function as minimized by the ILP described in Subsection 6.1.2 takes into account execution overhead to rule out unpromising parallelization candidates. If it lacks runtime information, however, the statically assumed default values are selected to favor parallelism. One example are function call execution times: if no runtime profiling information is available for a specific call-site, the called function is assumed to run long enough to outweigh potential parallelization overhead. This rule also applies in case of indirect function calls. A similar example are loop trip counts: if unavailable, iteration counts are assumed to be high enough to allow profitable parallel execution.

   This behavior is deliberately chosen to not miss an opportunity. It requires however an efficient runtime system to effectively evaluate and rule out unprofitable candidates. Information about such candidates is persisted in the filesystem of the executing machine to minimize the likelihood of the runtime system to repeatedly try and fail to profitably parallelize the same candidates during future executions of the application.

Furthermore, the information stored during this runtime-adaptive process is taken into account, if available, during future recompilation and static re-evaluation of parallelization candidates.

In this section, we describe how the statically found parallelization candidates are enriched, combined, evaluated and just-in-time compiled within the $ParA_\gamma$ runtime system.

2. It is of general nature: statically, $ParA_\gamma$ seeks to exploit the parallelism of an application as such. Neither does it take into account, nor fix a specific implementation of the parallelism at runtime. Consequently, it can only take into account inevitable cost that is not specific to a particular implementation. In order to exploit the parallelism in the most profitable way, or profitably at all, the $ParA_\gamma$ runtime system has to "specialize", or optimize the parallel code. Just like many other well-known compiler optimizations do on sequential code, such optimizations working on the parallel IR need to take into account features of the program and possibly also the execution platform that support efficient parallel execution. Such features include a statically known, or, statically unknown, but at least provably loop invariant iteration range, the variadicity and type of reduction locations, or the recursion scheme of parallelization candidates, for example.

   Loop blocking (Section 7.3) is one very important optimization of the parallel code; parallel section propagation (see Subsection 4.1.4) is another such optimization. *Runtime adaptive dispatch* (Section 7.4) is an important feature of a parallel runtime system. *Loop blocking* and *runtime adaptive dispatch* are described further in this section.

7.1 Runtime Profiling

The static scheduler of $ParA_\gamma$ decides for parallel execution if in doubt and relies on the runtime system to take all available information into account to select only the profitable, or at least drop erroneously selected unprofitable candidates. Consequently, the runtime

system needs to collect the relevant information, most importantly profiling data. In particular, the $ParA_\gamma$ runtime system collects two kinds of profiles:

- *Call-site execution times*, which is basically a context-sensitive form of average method execution time.
- *Branch profiles*, used to estimate the execution cost of decision nodes and to derive loop trip counts.

7.1.1 Call-site Execution Times

This section describes the details of *Sambamba*'s call-site profiling capabilities[1]. As mentioned earlier, the main purpose of collecting execution time profiling information in *Sambamba* is to enable optimizations to estimate the size, i.e., execution time, of certain pieces of code in order to estimate the profit expected to be achievable by parallelization. In particular, the parallelization overhead introduced to the critical path of execution needs to be outweighed by the execution time of code removed from it. A big contribution to the execution time of a given piece of code, apart from long running loops, typically stems from called functions. Their execution time depends on many features only available at runtime, such as input values and size, or system load, and is theoretically impossible to compute statically. Consequently, we have to resort to collecting runtime profile information. Such information at least provides insight into previous runs, and, in many practically relevant cases, allows to make conclusions about future instances, which are precise enough to allow for qualified parallelization decisions.

To gather information about the execution time of a call instruction, two options exist, which are both supported by the profiler implementation of *Sambamba*: either the potentially called function is instrumented to collect execution time profiles, or the call instruction itself is instrumented. $ParA_\gamma$ uses the latter option for the following reasons:

- Instrumenting the interesting call-site only instead of the called function is often times the only viable option, for instance in case of indirect function calls (e.g.,

[1]The conceptual work of the call-site profiling capabilities of *Sambamba* has been done by Clemens Hammacher and myself. The final implementation in C++ has been mostly done by Clemens Hammacher.

calls to dynamically dispatched C++ methods), for which the called function is not statically known or even varies dynamically. In case of external function calls to a dynamically linked (binary) library it is the simplest yet most precise approach.

- It keeps the overhead on non-candidate call-sites minimal. Instrumentation of the called function would introduce the profiling overhead to calls from regions which are not subject to parallelization. Alternatively, a separate profiling copy of each function is kept and the candidate call-sites rewired to those copies. Again, this renders difficult in case of indirect or external function calls.

- The information collected by call-site profiling is more precise in the sense that it adds context-sensitivity. Imagine a hypothetical *sort* function sorting arrays of data; the execution time when being called from within a routine sorting a simple manually crafted address book might significantly deviate from the time it takes to sort the data usually involved in bigger problems. Consequently, parallelizing the processing of a simple address book on a mobile phone might not be profitable, while doing so in the context of processing large data-sets certainly can be. The context-sensitivity provided by call-site profiling, in contrast to function profiling, is able to distinguish both instances.

- *Sambamba* employs dynamic recompilation, frequent exchange of function versions in the running system, and flexible parallel/sequential execution (see Section 7.4). Instrumenting the function instead of the call-site, and in particular keeping an instrumented and an uninstrumented version of each potentially called function would significantly increase the complexity and the overhead of instrumentation and just-in-time compilation.

Functions subject to potential parallelization, i.e., those functions in which the static parallelizer found promising parallelization candidates, are instrumented at runtime to collect the average execution times of contained call-sites.

One technical but important detail in the context of execution time profiling in *Sambamba* is the necessity to ignore invocations of a function which is lazily just-in-time compiled

during that invocation. If at least one function on the call stack below a profiled call-site is just-in-time compiled during the profiled invocation, the gathered information is called *poisoned* and consequently ignored, as it might otherwise significantly influence the precision of profiling.

7.1.2 Branch Profiles

Apart from the execution time of call-sites, another very important information to enable the estimation of parallelization profitability is branch profiles (or execution frequencies), from which also loop trip counts can be derived.

$ParA_\gamma$ uses the PDG (see Subsection 5.1.1) as its central program representation throughout the analysis and transformation process, also during profitability estimation. What we are therefore interested in is the execution frequency of the individual PDG nodes (instructions or basic blocks): if we know how often a specific node is executed, and how long it takes to execute (its size, see Subsection 6.1.1 and Subsection 7.1.1), we can, by weighted accumulation, derive the expected average execution time of the whole PDG, i.e., the function it represents. Furthermore, we can compute the fraction of a function's execution time spent executing a specific PDG node or code region, like a loop for instance.

To collect the relevant information at runtime, $ParA_\gamma$ instruments the code during sequentialization (see Subsection 5.1.2) of the PDG. To keep code generation overhead low, and data management and mapping to the PDG at runtime simple, the program locations to insert code to increment respective frequency counters are chosen to correspond to (non-empty) PDG group nodes. From those, the execution frequency of each instance of the three PDG node types can be derived: group nodes are directly captured, and the execution frequency of a regular or a decision node corresponds to the sum of the execution frequencies of its parents in the control-dependence sub-graph of the PDG, which in turn are group nodes.

Note that the instrumentation locations (and, equivalently, frequency counters) chosen this way directly correspond to the distinct control conditions of a function. The number of necessary locations/counters is strictly smaller than for the frequently used practice to

instrument all control-flow successors of branching instructions, and an order of magnitude smaller than for instrumenting all basic blocks. It nevertheless is not minimal as shown by Knuth and Stevenson [110] or Ball and Larus [111] who prove a placement based on a maximum spanning tree of the CFG minimal. The key idea, enabled by Kirchhoff's first law[2], is to compute certain frequency counters based on other counters they depend on.

We decided not to implement the optimal approach for a few reasons:

- While the edge frequencies of CFGs respect Kirchhoff's first law, the edge frequencies of control dependence graphs do not, which makes the approach not directly applicable to the situation at hand.
- Computing the optimal placement, in particular the spanning tree, significantly increases the instrumentation time as shown by Ball and Larus [111]. While the authors argue that the overhead is quickly outweighed by the reduced execution time and smaller amount of required counters, this is not as relevant in our situation: the low number of necessary profiling runs (cf., Edler von Koch and Franke [112] and Subsection 7.1.3) as well as the necessary storage to keep the counter relations reduce the expected reduction of overhead.

Practically speaking, the sufficiently low overhead of the chosen approach did not justify the increased complexity of the implementation as well as the instrumentation at runtime.

An important measure to reduce profiling overhead in the context of parallel execution however is to privatize the frequency counters. Each thread gets assigned a private copy of each frequency counter, whose allocation is aligned with the cache line size. On a requested read of the counters for a PDG, which happens for instance on reassessment of parallelization decisions at runtime, the private counters are collected and accumulated.

[2]Kirchhoff's first law, also known as *Kirchhoff's Current Law*, originates from the field of electrical engineering and describes the conservation of electrical charge: at any junction in a circuit, the sum of the currents arriving at the junction equals the sum of the currents leaving the junction.

7.1.3 On Profiling Overhead

In order to minimize the introduced runtime overhead, profiling is only enabled for relevant code parts: only call-sites contained in statically determined candidate regions of parallel execution are profiled. Branch profiles are only collected for functions containing at least one parallelization candidate.

Theoretically, not every single invocation of a target function needs to be profiled. If enough samples are already available and profiling information is stable, the likelihood to learn something fundamentally new is low and profiling could be disabled with an increasing probability. Edler von Koch and Franke [112] have shown that profiles, in particular data dependence and control flow related ones, stabilize quickly.

The adaptive parallel code generation of $ParA_{\gamma}$ and *Sambamba* in its current implementation nevertheless keeps gathering execution time profile information during the whole program execution with a fixed, but configurable sampling rate (in profiled invocations per second). This is done in order to be able to flexibly react to changing profile information, for instance after parallelization and exchange of a called function. Branch profiles, which only in very rare cases are affected by the system load, parallelization or any other transformation performed by $ParA_{\gamma}$ do usually not need to be collected any more, once they have stabilized. They can however be affected by changes in input values and size.

In its current implementation, $ParA_{\gamma}$ does not automatically decide when to collect branch profiles and when to stop doing so. Binaries produced with *Sambamba* and $ParA_{\gamma}$, instead provide a runtime parameter to enable gathering of branch profiles in dedicated profiling runs of the application. Profiles are collected and persisted for use in the ongoing and later executions of the application, as well as during future re-compilation.

Furthermore, significant engineering effort went into minimizing the overhead of profiling to an almost negligible fraction of the relevant functions' execution time. Functions for which this does not apply are most likely too short running to be profitably parallelizable. After observing a configurable amount of very short running invocations, corresponding function calls are stopped from being profiled, a short execution time is assumed by relevant cost functions.

7.2 Candidate Composition

Under certain conditions, the runtime parallelizer of $ParA_\gamma$ is triggered to reassess parallelization decisions made earlier during a program run. The most important of such triggers is a significant change in profiling information. If, for instance, the average execution time of a call-site contained in a parallelized function F changes significantly after parallelization of the called function G (or the opposite: switch from parallel to sequential execution of G), reassessment of the parallelization decisions for F is advisable.

The parallelization decisions of the $ParA_\gamma$ runtime system are based on the statically identified (local) parallelization candidates (or schedules). Remember that one local parallelization candidate corresponds to a parallel schedule for a group node, i.e., for a set of PDG subgraphs executed under the same control-condition, represented by the respective group node. For each group node, the static parallelizer finds at most one parallel schedule. That means, a PDG PDG_F for a function F containing n non-empty group nodes can have a set Γ_F of up to n associated parallelization candidates $\gamma_1 \dots \gamma_n$, out of which the runtime system has to select a combination to compile a parallel version F_π of F.

To evaluate the expected performance gain for a combination, $ParA_\gamma$ first ranks the candidates according to their expected individual contribution. This is done by evaluating the expected execution cost of F (see Subsection 7.2.1) assuming parallel execution of each individual candidate in turn and comparing it to the expected sequential execution time.

In the following we assume the candidates $\gamma_1 \dots \gamma_n$ to be sorted in decreasing order of their individual contribution, i.e., γ_1 is the candidate with the biggest expected individual saving of execution time, γ_n the candidate with the lowest expected saving.

We call a local candidate γ_x *realized* if it is selected for parallel execution.

We denote by $F[\Delta]$ with $\Delta \subseteq \Gamma_F$ the global schedule for F with the local parallel schedules contained in Δ realized. All local schedules not contained in Δ are not realized, their corresponding PDG group nodes are scheduled sequentially. $F[\gamma_x, \gamma_y]$ denotes the global schedule for F with γ_x and γ_y realized.

Note that the evaluation of the expected execution time of F already takes into account the available and potentially changing profile information and can therefore not be statically pre-computed.

Based on the ranking of local parallel schedules, the expected execution time saving of all combinations of the m most promising candidates $\gamma_1 \dots \gamma_m$ are evaluated. The number m of candidates to take into account per function is configurable; it defaults to 3, i.e., for each function, eight combinations of the three most promising candidates are evaluated. The best combination is selected to finally generate the best version of F, which might very well be the sequential one.

Note that the selected combination might contain nested parallelism if one parallelized group node g_2 is reachable in the PDG from another parallelized group node g_1. Or, equivalently, the control condition represented by g_2 implies the one represented by g_1.

The selected global schedule $F[\Delta]$ for the containing function F is used during PDG sequentialization (see Subsection 5.1.2) to compile a *ParCFG* for F. Also during sequentialization, the necessary code generation for realized reductions (see Subsection 5.2.1) and privatization (see Subsection 5.2.2) takes place. On the *ParCFG* level further (optional) transformations like for instance dynamic blocking (see Section 7.3), parallel section propagation (see Subsection 4.1.4), and other control-flow and parallelism specific optimizations and transformations are performed before producing the final parallelized control-flow graph. Before just-in-time compiling this CFG to the binary level and installing it into the running system it is equipped with the necessary dispatching mechanisms (see Section 7.4).

7.2.1 Execution Cost Evaluation

The expected execution cost of a function F under a global schedule $F[\Delta]$ is computed in a post-order traversal of the control dependence sub-graph of its PDG PDG_F and corresponds to the accumulated cost of the PDG's dedicated root node. Remember that the accumulated cost, which we are interested in, is not the execution cost of any given PDG node itself, but instead the cost of the reachable PDG-subgraph.

The execution cost computation at runtime almost directly corresponds to the estimation at compile-time as described in Subsection 6.1.1. It is computed for each of the three node types as follows:

Regular Nodes. The accumulated cost $\|r\|_{dyn}$ of a regular PDG node r is equivalent to the static version: Assuming PDG nodes represent basic blocks of instructions, the cost of a node is determined by summing up the estimated cost of the instructions it contains. Again, the cost of instructions accessing memory, like *reads* and *writes* for instance, depends on the size of the accessed memory region (as far as statically known); The cost of *call* or *invoke*[3] instructions is estimated based on available runtime profiling information as described in this chapter.

Decision Nodes. Also nearly equivalent to the statically computed estimation, the accumulated size of a decision node d is computed as the frequency-weighted sum of the accumulated sizes of the PDG nodes which are control-dependent on d:

$$\|d\|_{dyn} := \sum_i (\|d[i]\|_{dyn} * freq(d_i)) + |d|_{dyn}$$

where $freq(d_i)$ is the frequency of d selecting child node $d[i]$ for execution, and $|d|$ is the non-accumulated size of d, which is computed in a similar way to the cost of regular nodes (remember that, just like a regular node, a decision node corresponds to a basic block; the distinguishing feature of a decision node is that in contrast to a regular node, it has multiple control-flow successors).

Frequencies are statically estimated or read from possibly available profiling information collected during the ongoing or earlier runs of the application.

One difference to the static computation of decision node execution time is the treatment of loop headers. Note that a loop header in the PDG is always a unique decision node on which the loop body statements are control-dependent. Also, note that the PDG loop header does not necessarily correspond to the CFG loop header. Nevertheless, it is also unique, even in the presence of multiple loop exit conditions

[3] An invoke instruction in LLVM is a call instruction which possibly throws an exception.

themselves introducing control-dependences: by definition, there is one loop exit block on which all others are control-dependent.

The only special case are static endless loops, whose headers have only one successor in the CFG and can thus by definition not impose any control-dependences (and are consequently no PDG decision nodes). This corner case is however technically ruled out by assuming the existence of an artificial additional control-flow successor of the CFG loop header. As a result, the loop body statements are control-dependent on the header.

The accumulated execution cost of a PDG loop header is computed at runtime like for any other decision node, multiplied by the loop-trip-count of the respective loop:

$$\|d\|_{dyn} := (\sum_i (\|d[i]\|_{dyn} * \mathit{freq}(d_i)) + |d|_{dyn}) * \mathit{ltc}(d)$$

where $\mathit{ltc}(d)$ is the loop-trip-count of the loop with PDG loop header d, or 1 in case d is no PDG loop header. The average loop trip count is computed based on the profiled execution frequencies.

Group Nodes. The accumulated execution time computation of a group node g at runtime is parametrized by a schedule γ, which can be either sequential or parallel:

$$\|g\|^{\gamma}_{dyn} := \sum_{s \in \gamma} \max_{t \in s} \sum_{i \in t} \|g[i]\|_{dyn}$$

This computes the sum of accumulated execution times of g's children on the critical path of schedule γ. Remember how a parallel schedule is subdivided in stages (s) and threads (t) as described in Chapter 6 and shown in Figure 6.1.

This definition also covers a sequential schedule which contains only one stage and one thread in which case the definition of $\|g\|_{dyn}$ coincides with its static counterpart $\|g\|$.

7.3 Dynamic Blocking

Dynamic blocking[4], another important feature of the $ParA_\gamma$ runtime system, aims at increasing the size of parallel tasks, as the overhead of enqueuing parallel tasks can easily outweigh the actual work to be done by the tasks. This frequently is the case for parallelized loops doing little work per iteration of the loop body, like for instance in the *BiCG* example shown in Figure 1.4a. The reentrant parallel section for such a loop contains exactly one reentrant task representing the iteration part of the loop, and thus also all loop-carried dependences. One or several non-reentrant tasks in the same section represent the actual parallel work to be spawned off for parallel processing in each iteration. If these tasks only contain a small amount of code, like loading a value from an array and performing a reduction operation, then the overhead of handling the parallel tasks nullifies the benefit of parallel execution.

Such a loop should however by far not be discarded from the parallelizer's perspective. Plenty of research in automatic parallelization has shown that the parallelism in such loops can successfully be exploited. Modern instances of *Decoupled Software Pipelining* [10, 74, 76–78] and the *Helix* [81, 82] family of approaches in particular have shown impressive performance improvements by effectively reducing the overhead of parallelization and the parallelism enabling necessary synchronization and communication mechanisms. Those and most other approaches of automatic parallelization however usually seek for program patterns suitable for the specifically chosen approach of parallelization. Other possibly parallel but non-fitting program parts are ignored, effectively ruling out parallelization candidates right away.

$ParA_\gamma$ comes from a different direction: not taking into account profitability and suitability in the first place, a very wide range of parallelism is found and exposed to the runtime system by the static parts of $ParA_\gamma$. The number of parallelization candidates found by

[4]Dynamic blocking has been excogitated and an early instance implemented by myself. The currently used, and by far improved, version of it has been devised by Johannes Doerfert and myself. The implementation has been mostly done by Johannes Doerfert.

$ParA_\gamma$ is therefore expected[5] to be way higher than for approaches seeking for special patterns of parallelism only. Candidates are then left for selection and composition as described in the previous sections to finally compile *ParCFGs* for relevant parts of the application. In order to be profitable in the above mentioned case of long-running loops with minimal work per body instance, $ParA_\gamma$ has to perform further, parallelism specific, optimizations on the *ParCFGs*. $ParA_\gamma$'s *dynamic blocking* is one very important example of such an optimization. It increases the task size of a reentrant parallel section (i.e., a parallel loop) by dynamically joining (blocking) a number of parallel tasks, before enqueuing the whole block as one batch task. This greatly reduces the number of small tasks doing negligible work and the associated overhead and pressure on the dynamic scheduler.

Apart from the direct effect of decreasing the $\frac{overhead}{work}$-ratio of small parallel tasks, it further allows to join necessary parallelization-specific per-task overhead: any task accessing privatized data, for instance a privatized reduction location, profiling counters, or simply privatized data, profits from blocking: the private copy needs to be determined only once per batch task; for atomic operations involved in reduction, we only need to update the shared location once.

Equally important is the reduced communication overhead and required storage: instead of computing a simple iteration variable in the reentrant part of a loop and communicating its value to all the spawned tasks, it is communicated not more than once per block of tasks, provided its computation is deterministic and free of side-effects. The computation code is replicated per block in that case[6], communication and stalling of tasks waiting for input reduced. This technique is not only applied to iteration variables. The results of side-effect free and deterministic computations of values consumed by a blocked task are not communicated, but instead recomputed within the consuming block. Loop-invariant values are communicated only once per block.

[5] We say "expected" because during the course of our studies none of the most promising approaches have been made available to us for evaluation, despite the fact that we asked for it multiple times. Reasons mentioned where the quality of code, ongoing refactorings, and assumptions to a hypothetical hardware (Helix for instance is simulated because of an assumed but non-existing inter-core ring-cache.).

[6] Replicating the loop control structure and necessary value computations is also an essential part of decoupled software pipelining.

Note that all the described measures are not necessary in the well-known and regularly applied (recursive) range splitting since they are included by design. In contrast to recursive range splitting, dynamic blocking is however more general and thus wider applicable because it does not require a statically known, and not even a loop invariant iteration range of the loop. A downside is that the thread that collects the tasks before spawning them in one block may become the bottleneck of parallel execution. Therefore, $ParA_\gamma$ applies another optimization: in case the loop iteration range is loop invariant, i.e., known before entering the loop, the $ParA_\gamma$ runtime system produces code that immediately distributes the loop execution equally among available threads, which basically corresponds to a one-level (i.e., non-recursive) range splitting.

Finally, the dynamic nature of $ParA_\gamma$'s blocking allows to arbitrarily change the block size at runtime in case the dynamic scheduler (or the executing system) is under- or oversubscribed. This is not currently done by $ParA_\gamma$, which instead allows to select the block size as a parameter to the parallelized binary, or automatically chooses a sensible default.

7.4 Adaptive Dispatch

After the $ParA_\gamma$ runtime system has selected the best combination of parallelization candidates in the light of current profiling information and the observed behavior of the application, parallel code is generated in a parallel intermediate representation. This representation is further optimized as described in the previous section; finally, a parallelized control-flow graph is generated by translation and augmentation of the *ParCFG*. This parallelized version is dynamically produced, based on dynamically gathered information; All parallelization decisions can be reassessed in the future and a fall back to sequential execution, in case the parallel version is deemed unbeneficial, is always possible. Nevertheless, the reassessment process as described above, followed by PDG sequentialization and parallel code optimization is not as flexible as is necessary for profitable exploitation of parallelism. What is needed, apart from flexible parallelization, is flexible parallel execution. The runtime system needs to be able to switch between parallel and sequential execution in reaction to the executing system's changing load, for instance, without immediately

dropping all parallelized functions. A temporary peek in the system load, caused by an independently running application in the same environment, should not cause a complete fallback to sequential execution, followed by a slow adaptation to parallel execution, once the system load has dropped again.

$ParA_\gamma$'s chosen solution is to introduce a dispatch mechanism, which is able to dynamically choose between parallel and sequential execution upon each individual invocation of a function F. By monitoring the execution of the program, as well as important properties of the execution environment, $ParA_\gamma$ chooses which of the versions to dispatch. Sensible criteria are based on the current system load (*load*), the nesting depth of parallel tasks (*tnd*), or the utilization of the dynamic task queues (i.e., the number of tasks in flight, *tif*). These criteria are described in the following subsections.

7.4.1 Load-based Dispatch (load)

The load based dispatcher is the most straightforward dispatch mechanism and has already been shortly described in Subsection 4.1.5. As mentioned, querying the system load inevitably requires a system call. It is therefore in almost no case profitable to query the system load every single time a decision for or against parallel execution needs to be made. Instead *Sambamba* employs a dedicated thread, polling the system load in a configurable interval defaulting to once every second.

While this mechanism is effective and the application resigns from parallel execution in phases of high system load, it imposes high overhead and reacts quite slowly due to the fixed polling interval.

For these reasons, load-based dispatch has been mainly superseded by *tnd* and *tif* dispatch, the latter of which also indirectly reacts to high system load caused by foreign applications, while imposing significantly lower overhead.

```
1  void mergesort(Element *A, Element *tmpA, long size) {
2      if (size < 2)
3          return;
4
5      long half = size / 2;
6      Element *B = A + half;
7      Element *tmpB = tmpA + half;
8
9      mergesort(A, tmpA, half);
10     mergesort(B, tmpB, size - half);  // execute in parallel
11
12     merge(A, half, B, size - half, tmpA);
13
14     memcpy(A, tmpA, sizeof(Element) * size);
15 }
```

FIGURE 7.1: Simple sequential *C*-implementation of *mergesort*.

7.4.2 Task Nesting Depth Dispatch (tnd)

Dispatching based on the nesting depth of parallel execution is of particular importance in the case of parallelizing recursive functions like for instance the *fft*-example in Figure 1.4b.

As a motivating example, and for the purpose of a detailed preliminary evaluation of dynamic dispatching, consider the sequential *C*-implementation of *mergesort* shown in Figure 7.1, in which the recursive calls to *mergesort* itself should be executed in parallel.

Figure 7.2 illustrates the recursion tree of *mergesort*. It is a straight-forward, but naive approach to excessively spawn parallel tasks at each recursion level, since the work done

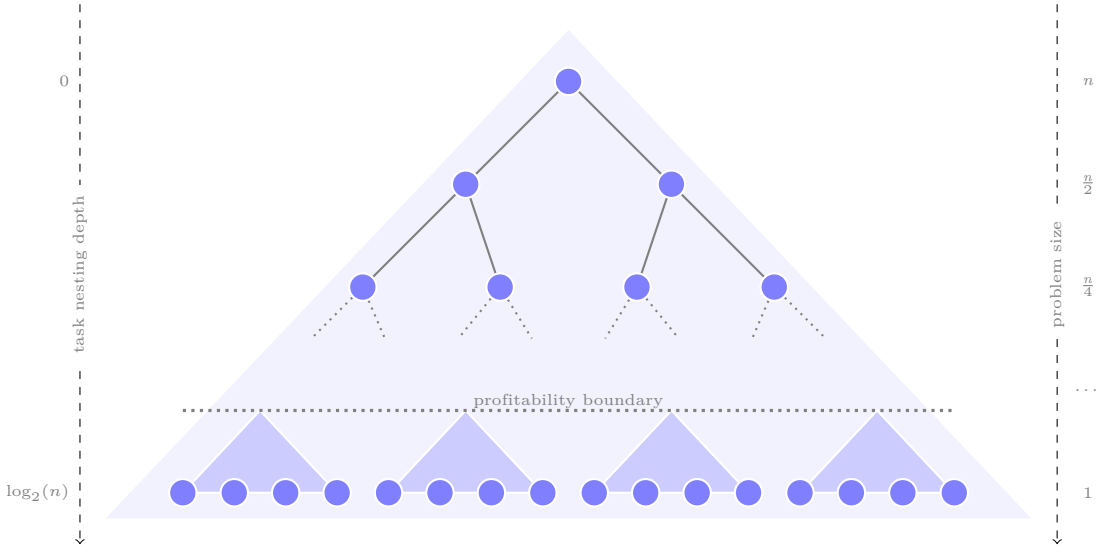


FIGURE 7.2: Recursion tree of *mergesort* with a profitability boundary based on problem size.

by each instance of *mergesort* decreases with every recursion step. Starting from a certain depth of recursion, once the size of the array to sort is too small, the overhead of parallel execution outweighs the actual work. Parallel execution of those calls towards the leafs of the recursion tree is not profitable. As the number of leafs in the mostly balanced binary recursion tree is half of the number of overall invocations, and parallelization is unprofitable not only at the leafs, only a small fraction of invocations of the brute-force parallelized version of *mergesort* is profitably parallelizable.

While parallelizing manually, this issue is frequently solved by introducing a parallel execution threshold—an artificial boundary, which strictly speaking has nothing to do with the algorithm as such, but instead is a mere artifact of the parallel execution environment. This is problematic for several reasons: the switch-over criterion of the boundary depends on the problem size and the parallel execution capabilities of the application's target execution environment. Determining an optimal value is thus not possible at compile-time and always has to be conservatively chosen. Furthermore, the boundary-related code increases the code complexity and reduces maintainability.

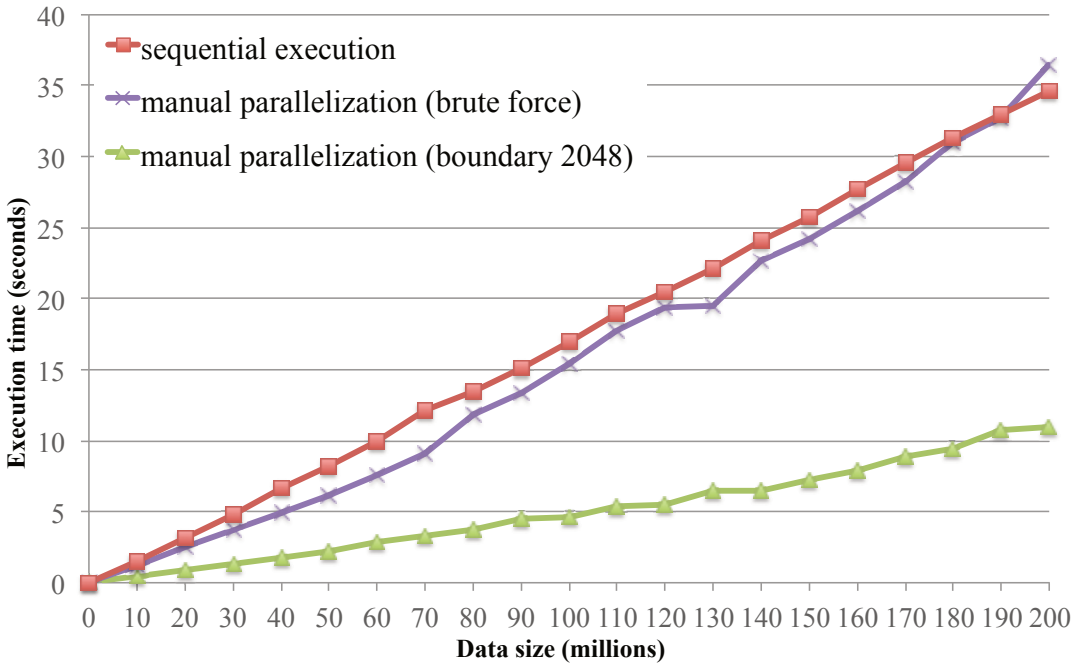


FIGURE 7.3: Execution time of *mergesort*: sequential vs. brute-force parallel vs. manual boundary at arrays of size 2048 (linear scales).

With respect to the performance of parallel execution, even a conservatively chosen array size of 2048 elements as switch-over point from parallel to sequential execution significantly improves the situation as shown in Figure 7.3. The x-axis shows the input size; the y-axis shows execution time (lower is better). Figure 7.4 shows the same data on a double log scale, emphasizing smaller input sizes to clearly see important features and the effect of the switch-over point. All experiments have been run on a quad-core desktop-class machine with hyperthreading. All parallelized versions are parallelized using Intel TBB [40], and thus use a state-of-the art work-stealing scheduler at runtime. Sequential and parallel versions are optimized and use the same (parallelization independent) compiler flags.

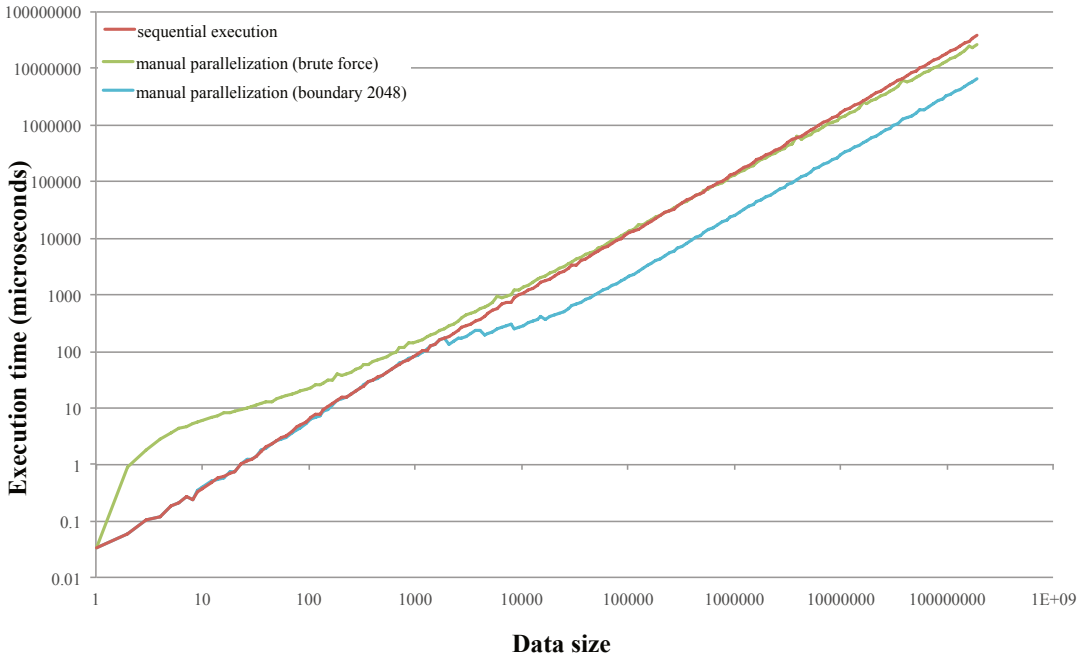


FIGURE 7.4: Execution time of *mergesort*: sequential vs. brute-force parallel vs. manual boundary at arrays of size 2048 (log scales).

Both figures clearly show that brute-force parallel execution is only slightly faster than sequential execution. For input sizes smaller than circa 30.000 elements it is not profitable at all and is easily outperformed by the sequential version of *mergesort*. Again, remember the log scale in Figure 7.4 showing that for arrays containing less than 1.000 elements the performance difference spans several orders of magnitude.

The third line shows the performance of the manually TBB-parallelized version of *mergesort* with an artificial boundary of 2048 array elements to switch to sequential execution. It is

clearly visible that this version stays sequential for very small inputs and unsurprisingly matches the sequential performance in the range from 1 to 2047 elements. In the range from 2048 to 4095 elements we see a two-fold speedup (remember the log-scale) which stems from the parallelization at the first level of recursion. The performance roughly doubles again when the second level of recursion is also parallelized in the range of 4096 to 8191 elements. At the third level, we see another big improvement, showing that hyperthreading can be quite nicely exploited in this case. The fourth level only shows a small, and last, performance improvement. Parallelization at deeper levels of recursion comes with no further performance benefits and only introduces overhead and increases the pressure on the dynamic scheduler.

As discussed earlier, 2048 has certainly been carefully chosen by an expert programmer; this choice however is based on experience, heuristics and at best measurements on a machine closely matching the final target machine. There is no reason to believe that this boundary is the best possible choice on all machines. Quite contrary, as clearly visible in Figures 7.3 and 7.4, on a machine having only 8 cores available for parallel execution there is no use in parallelizing more than three to four levels of (balanced) recursive execution. By selecting the switch-over boundary based on the dynamically available compute resources instead of statically estimated profitability estimates, $ParA_\gamma$ conceptually replaces the profitability boundary by a resource utilization boundary (Figure 7.5). This increases the task size for most inputs and effectively reduces the $\frac{overhead}{work}$-ratio.

The *tnd* dispatcher of $ParA_\gamma$ switches to sequential execution, once the nesting depth of parallel tasks exceeds $\log_2(\#Cores) + 1$. Note that this conservatively assumes a binary only recursion tree which produces more tasks than compute cores are available on the execution platform. Both is in favor of parallel execution and added flexibility of the dynamic scheduler.

In addition to at most one parallel version F_{par} of a function F, $ParA_\gamma$ keeps at each point in time the sequential version F_{seq} of F which corresponds to the purely sequential version of F and basically matches a statically compiled, unparallelized version. F_{seq} will never try to call anything else but the sequential version G_{seq} of any possibly called function G.

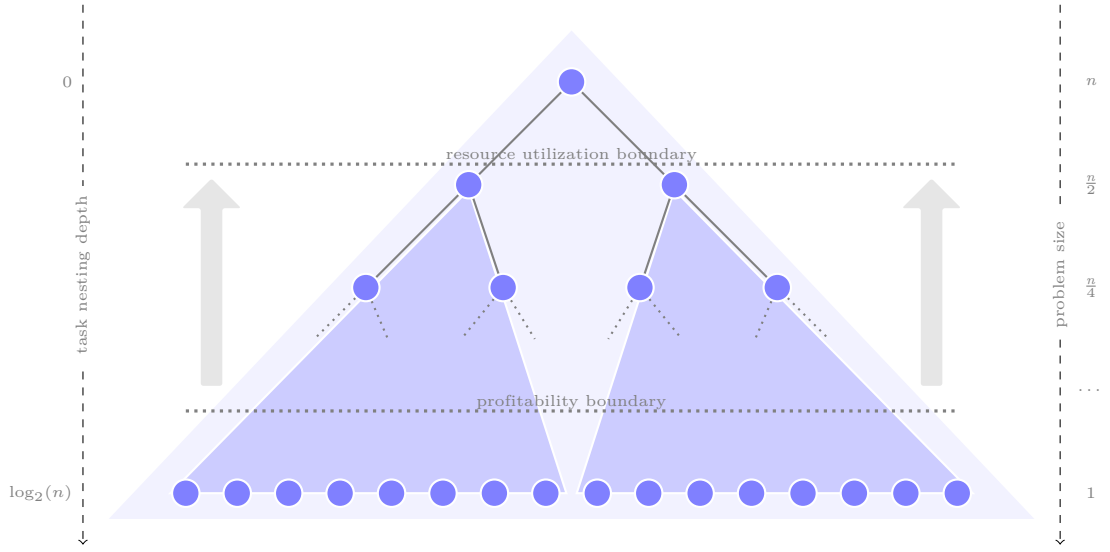


FIGURE 7.5: Recursion tree of *mergesort* with a resource utilization boundary based on the task nesting depth and available compute resources.

Once such a version of a function is called, execution will remain sequential, until this call returns. Consequently, all overhead of flexible parallel execution is removed.

In the case of the *tnd* dispatcher switching to sequential execution of F, F_{seq} is invoked. This decision is permanent for the deeper nested recursive calls as obviously, the nesting depth of parallel execution will remain the same and does not need to be reevaluated. There is no use in introducing the additional overhead.

Figures 7.6 and 7.7 show the effect of dispatching between parallel and sequential execution based on the nesting-depth of parallel execution in linear and logarithmic scales respectively. While the former one suggests that *tnd* dispatching slightly outperforms the version with a manually selected parallelization boundary for larger input, the latter figure shows that the *tnd* dispatched version clearly extends the range in which parallel execution is beneficial. On our evaluation machine, the boundary of 2048 array elements indeed is not the best choice. Also, note that the automatically parallelized and *tnd* dispatched version of *mergesort* does not require any parallelization related code to be written by the programmer. The code used in the experiment matches exactly the one shown in Figure 7.1 and thus only contains the algorithm-specific code.

As you can see in Figure 7.7, the *tnd* dispatched version unfortunately, but expectedly, shows the same bad performance as the brute-force parallel version for input sizes that

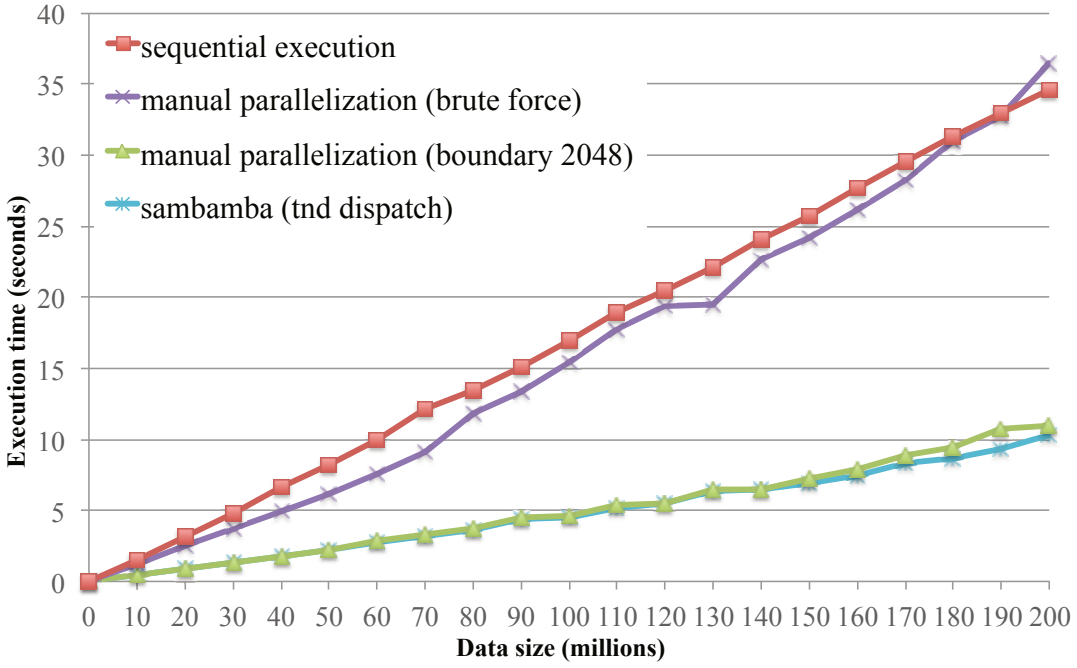


FIGURE 7.6: Execution time of *mergesort*: sequential vs. brute-force vs. boundary vs. tnd dispatch (linear scales).

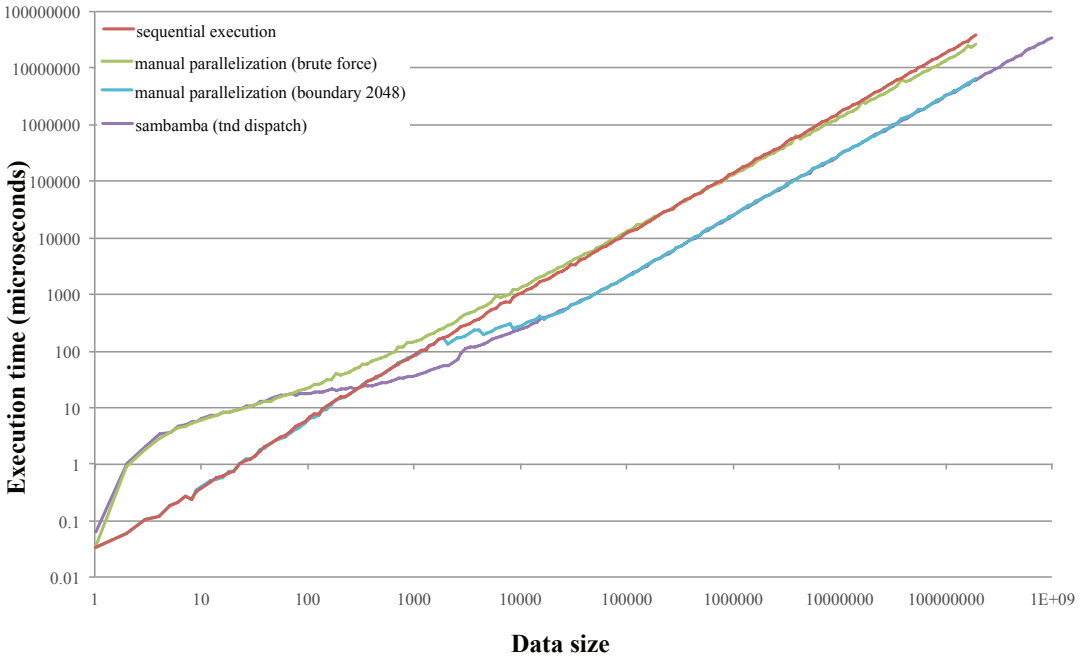


FIGURE 7.7: Execution time of *mergesort*: sequential vs. brute-force vs. boundary vs. tnd dispatch (log scales).

do not reach the nesting depth at which the dispatcher would intercept. Dynamic recompilation and function exchange based on collected profiling information is able to improve the situation as Figure 7.8 suggests. This version however is based on manually tuning relevant parameters of $ParA_\gamma$ based on properties of the *mergesort* application to enable the precision of the input range for sequential execution as automatically chosen by $ParA_\gamma$. Therefore, the performance numbers shown in Figure 7.8, although achieved in a fully automatic way by the final tuned implementation, should be understood as a feasibility study. Automatically tuning those parameters is possible (using the approach of Karcher and Pankratius [92], for instance) but left for future work.

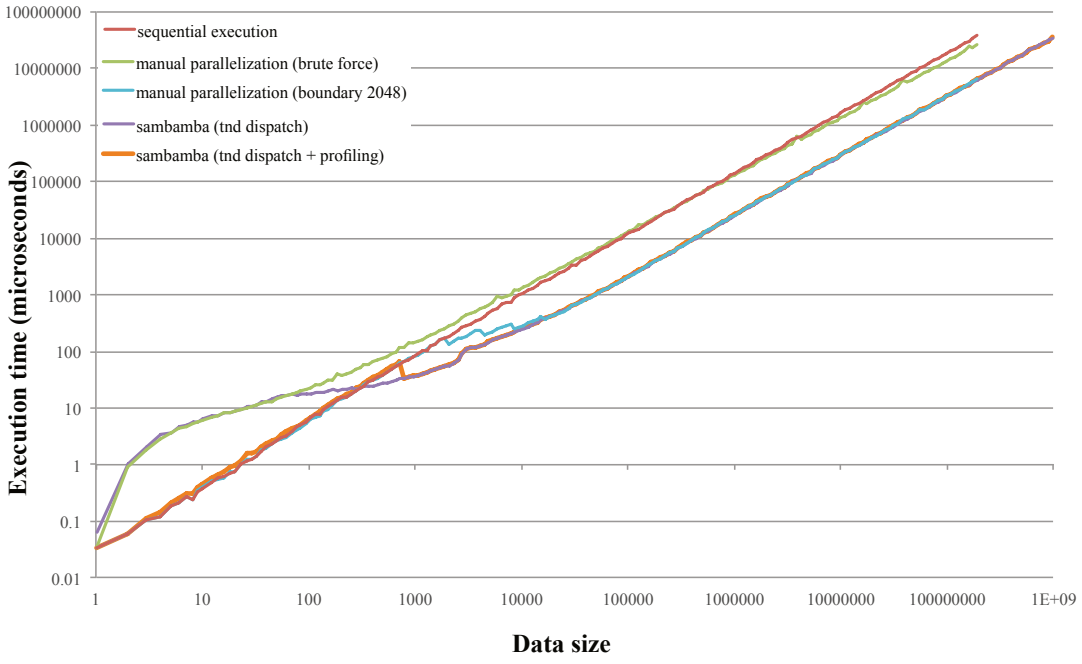


FIGURE 7.8: Execution time of *mergesort*: sequential vs. brute-force vs. boundary vs. tnd dispatch plus adaptive parallelization (log scales).

Apart from the profitability of parallel execution, Figures 7.6 to 7.8 further show the almost negligible overhead introduced by the dispatcher.

7.4.3 Tasks In Flight Dispatch (tif)

Switching from parallel to sequential execution based on the parallel task nesting depth is of course only effective in case of recursive or at least nested calls to parallelized functions.

As many, in particular general purpose applications do only rarely contain such calls, a different approach has to be chosen.

Like most modern parallel execution environments, $ParA_\gamma$'s runtime system uses a work-stealing scheduler based on a pool of threads executing tasks taken from multiple thread-local and one global task queue. We call the tasks put into one of the task queues and waiting for execution *in flight*. The *tif* dispatcher of $ParA_\gamma$ switches from parallel to sequential execution based on the number of tasks in flight and the compute resources available in the execution environment: in case the task queues contain a number of tasks exceeding twice the number of compute cores available in the executing system, $ParA_\gamma$ assumes the system to be oversubscribed and resorts to sequential execution.

Unlike the task nesting depth, the number of tasks in flight might decrease again for future calls to parallelized functions. It is therefore not desired to permanently switch to sequential execution once a high number of tasks in flight has been encountered. In addition to the parallel version F_{par}, and the always sequential version F_{seq} of each parallelized function F, $ParA_\gamma$ therefore keeps an additional sequential version F_{seq1} called when the *tif* dispatcher decides for sequential execution. F_{seq1} is a sequential version of F, which for any possibly contained (potentially recursive) call to a function G (in case of a recursive call $G = F$) calls the version of G which at the time of executing the call is currently installed in the running system. This might be a forced parallel version of G, a sequential version, or anything in between.

Figure 7.9 compares the performance of the *tif* dispatched version of *mergesort* to the versions discussed earlier. As can be seen, it closely matches the performance of the *tnd* dispatched and manually crafted versions. Despite the fact that dispatching based on task nesting depth is particularly well suited for balanced recursive algorithms like *mergesort*, certainly better than based on the number of tasks in flight, the *tif* dispatcher is a viable and profitable option. In case of programs containing no nested parallelism, it is the only effective option.

Note that the dispatcher based on the number of tasks in flight supersedes the expensive *load* based dispatcher as the number of tasks in flight indirectly reflects the load of the

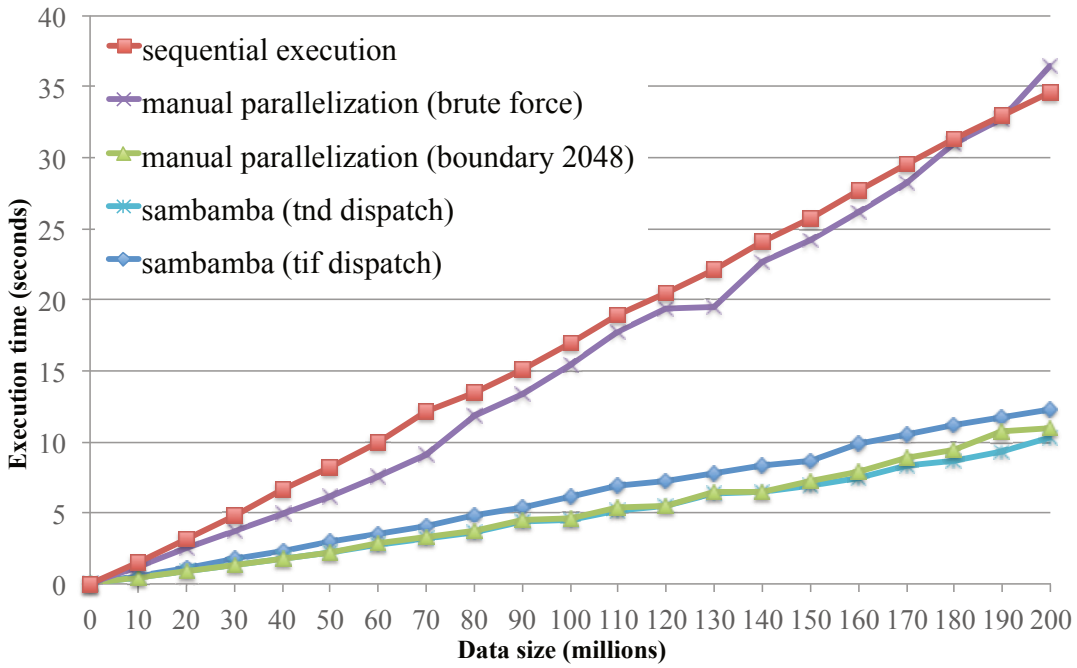


FIGURE 7.9: Execution time of *mergesort*: sequential vs. brute-force vs. boundary vs. tnd dispatch vs. tif dispatch (linear scales).

system. This is in contrast to dispatching based on the parallel task nesting depth, which is completely independent of the system load.

7.4.4 Combined Dispatch

So far the $ParA_\gamma$ runtime system can use one of three dynamic dispatch mechanisms to switch to the sequential version of a function. These dispatchers switch based on the following conditions:

- The system is already more than 90% utilized (*load*),
- more tasks than two times the number of cores available in the system are waiting for execution in the task queues (*tif*), or
- the nesting level of parallel execution is higher than $\log_2(\#Cores) + 1$ (*tnd*).

A parameter to the parallel binary compiled with $Sambamba/ParA_\gamma$ allows to control the selection of a specific dispatch mechanism. Leaving the expensive *load* based dispatcher

aside, the user is left with two sensitive choices, but almost certainly no knowledge of the internal structure of the application, which is needed to make a qualified decision.

A seemingly simple automatic approach would be to analyze the program and to use *tnd* dispatch for (transitively) recursive functions and those possibly containing nested parallelism along at least one path of the call-graph, and *tif* dispatch for all other cases. This however is difficult for two main reasons: facing indirect function calls the presence of nested parallel execution can only hardly be anticipated statically; the $ParA_\gamma$ runtime system might dynamically exchange the functions, possibly introducing or removing nested parallel execution, which would invalidate the choice of the dispatch mechanism for functions not exchanged.

$ParA_\gamma$ chooses a simpler yet effective approach: as the performance numbers shown earlier suggest a very low overhead of dispatching, using both, *tnd* and *tif* dispatch in combination seems affordable. Left with no manual decision, $ParA_\gamma$ defaults to a combined dispatcher which first determines the task nesting depth and switches to the always sequential execution if it exceeds the sensitive boundary. Otherwise it decides based on the number of tasks in flight. This combined dispatching effectively behaves like the *tnd* dispatcher on an under-subscribed system executing nested parallel code, and like the *tif* dispatcher otherwise.

This chapter introduced $ParA_\gamma$'s capabilities of runtime adaptive parallelization and parallel execution, which are the key to the efficiency of our generalized task parallelization.
Call-site execution time and execution frequency profiling form the basis of $ParA_\gamma$'s parallelization candidate composition and instantiation following the same optimization goal as the costly integer linear optimization performed at compile-time. Dynamic blocking effectively decreases the $\frac{overhead}{work}$-ratio and allows for efficient parallel execution of loops with short running bodies. Different methods of runtime adaptive dispatch take into account environmental properties like the over-subscription of the system or parallel task nesting to flexibly and quickly react to changing conditions.

Chapter 8

Implementation

In this chapter, we conceptually introduce and explain implementation details and decisions made in order to provide further insight into the nature of the *Sambamba* framework and $ParA_\gamma$, which is implemented as a module based on *Sambamba*. The purpose of the following sections is to foster future reuse and extension of *Sambamba*, which forms an essential part of the contribution of this thesis. *Sambamba*, as well as $ParA_\gamma$, have been carefully designed and crafted with modularity and extensibility in mind.

The goal of this chapter is not to introduce technically challenging, but conceptually uninteresting details. Instead it concentrates on details, that would have distracted the reader from grasping the main ideas of *Sambamba* and $ParA_\gamma$ if explained in detail in earlier Chapters. For real implementation details, we encourage the interested reader to study the source code as well as the contained documentation, which we are happy to hand out, together with our testsuite and benchmark data, to every interested researcher.

8.1 The *Sambamba* Framework

The *Sambamba* framework has been co-designed and developed with the parallelization modules $ParA_\tau$ and $ParA_\gamma$, as well as the speculation mechanisms described in thorough detail in [29]. Consequently, many of the design decisions have certainly been made in that light, which has influenced the selection of functionality provided by *Sambamba*.

The framework itself has, however, been kept clean from any parallelization artifacts, which are completely encapsulated in the corresponding *Sambamba modules*, which form the foundation of *Sambamba*'s extensibility. However, choosing automatic parallelization as the driving force of developing a framework like *Sambamba* has certainly not been disadvantageous for the available functionality. Automatic parallelization is one of the most invasive program transformations that a typical (automatically parallelizing) compiler would perform. It involves big parts of the tool belt available in a compiler framework.

In this chapter, we give a technical overview of the framework itself, followed by a description of *Sambamba* modules as the mechanism to extend *Sambamba*. Finally, we explain how *Sambamba* guarantees, at compile-time as well as at runtime, the isolation of modifications to the running system as performed by different concurrently running *Sambamba* modules.

8.1.1 Technical System Overview

Figure 8.1 provides a technical overview of the *Sambamba* framework itself. The figure concentrates mostly on the static parts; The dynamic parts are kept very lightweight as their behavior and usage heavily depends on the dynamic components of the linked *Sambamba* modules.

In principle, *Sambamba* behaves like a regular compiler/linker toolchain: it takes as input the application sources and produces an executable binary. Note that while in the figure we have chosen to start with *C/C++* sources as input for the sake of illustrating the complete toolchain, the real input to *Sambamba* is the linked *LLVM Bitcode module*. It is therefore in principle possible for *Sambamba* to handle programs compiled from any of the diverse set of input languages supported by the frontends available in the LLVM compiler infrastructure [95]. Development and testing has so far been done only with *C/C++* applications, however. Due to the absence (or, for the sake of correctness, the only explicitly stated presence) of obvious side-effects, it seems natural to try and parallelize a functional program, for instance one written in the *Haskell* programming language. This however imposes severe difficulties of technical nature: the Haskell frontend, for instance, weaves in and links a very complex runtime system including garbage collection, lazy

evaluation and the like, before producing the LLVM Bitcode that *Sambamba* gets to see. Domain and language specific optimizations, to which also effective parallelization would belong, have to take place before and are not usefully done afterwards.

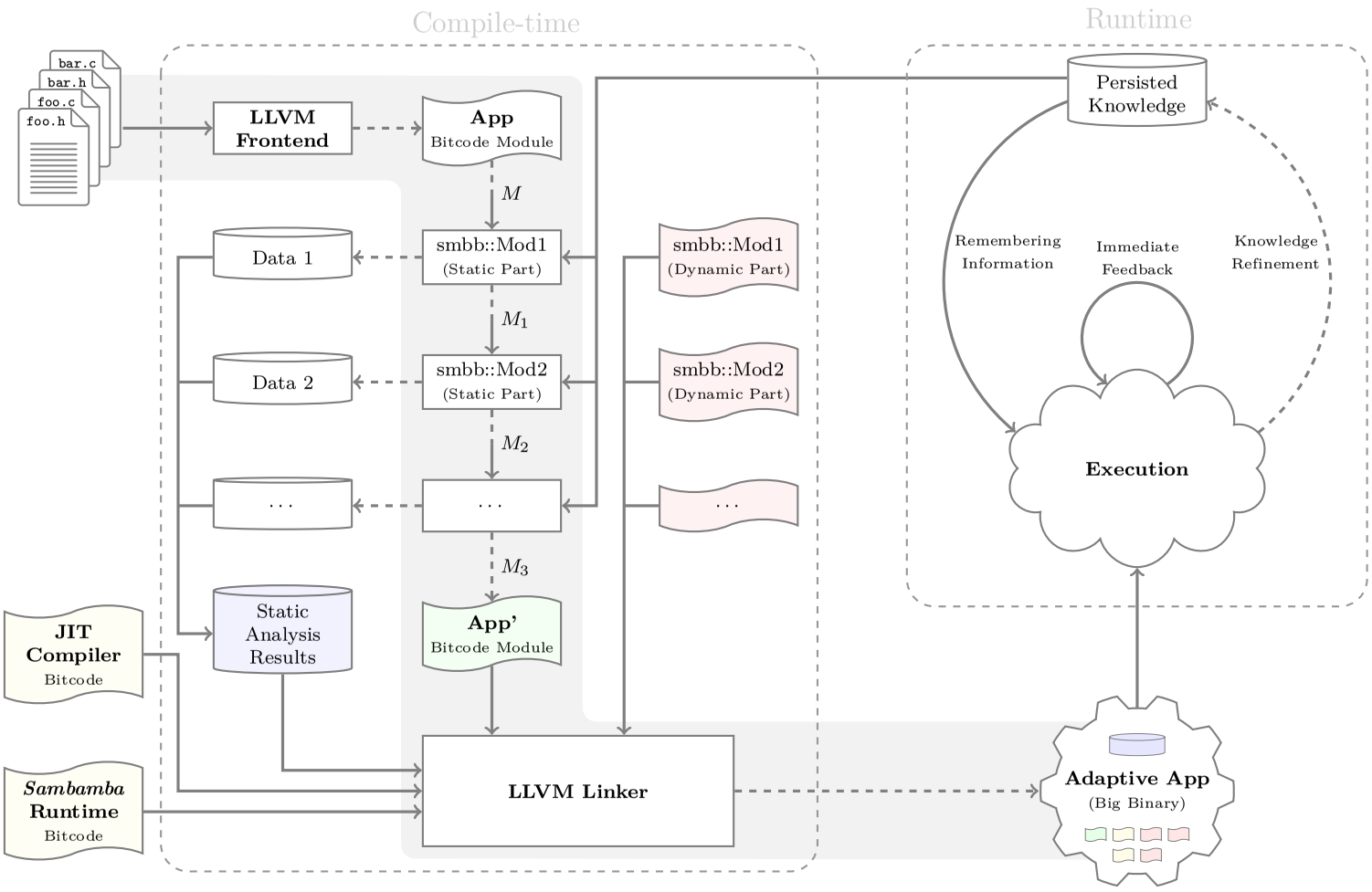


FIGURE 8.1: Technical overview of the flow of data and information through the *Sambamba* framework, independent of parallelization. This figure complements Figures 4.1 and 5.1 by providing a technical overview of the framework itself.

While we emphasize the language agnostic nature of *Sambamba*, this should not be understood as the capability to effectively parallelize or even usefully optimize every output produced by any LLVM compiler frontend.

The application bitcode is processed by a pipeline of regular compiler optimizations, as well as the static parts of *Sambamba* modules registered with the system. It is common in a well-designed compiler framework like LLVM to allow for extensibility by implementing and registering multiple isolated compiler passes. The difference between a compiler pass and a *Sambamba* module is, however, that a *Sambamba* module consists of a static and a dynamic part. The static part behaves completely like a regular LLVM compiler pass and can be registered and hooked into different stages of the static *Sambamba* compilation process. The dynamic part, however, is linked into the binary and invoked by the runtime system to allow for observation and modification of the running application.

After being processed and possibly modified by all registered *Sambamba* modules, the application bitcode is linked, together with the dynamic module parts, the *Sambamba* runtime system, a just-in-time compiler, and the statically computed analysis information into an *adaptive binary*.

At runtime, the dynamic parts of linked *Sambamba* modules can profit from observing the running application instead of having to completely rely on static information or information collected in potentially unrealistic predetermined profiling runs. By reacting to immediately available feedback while the application is running, the system can be adapted to changing environmental conditions, like changed system load, or user input, for instance.

Information collected during earlier executions of the binary is persisted on the executing system and stored in a permanently refined knowledge database. This is important to not having to start learning, for instance on the unprofitability of certain (speculative) parallelization candidates, or optimization parameters, from scratch each time the application starts executing. It is expected that, depending on the performed optimizations, during early executions the system might very well spend large fractions of the execution adapting to newly observed conditions until it finally stabilizes. The purpose of persisting

the learned information is to lower this fraction with every execution, and with it the overhead of adaption.

Additionally, the information learned during real (and thus realistic) executions is also available to the static parts of registered *Sambamba* modules during later recompilation of the application on the same system. Technically it is possible, but not done by *Sambamba* in its current implementation, to submit learned information to a central online database to be used by different users and during compilation on a system different from the one executing the final application. This is similar to submitting the (intermediate) results of the linear optimization performed by $ParA_\gamma$ during static compilation (see Subsection 8.2.3).

8.1.2 Sambamba Modules: Compile-time vs. Runtime

Sambamba modules consist of two parts: a static part being active at compile-time, and a dynamic part being active at runtime, while the application executes. Both parts are optional: it is not required for a *Sambamba* module to act during compilation. Neither is it forced to have a dynamic component, although this is certainly the main purpose and defining feature of a typical *Sambamba* module.

The static part is invoked during compilation and closely resembles a *LLVM ModulePass*[1]. It can make use of the regular mechanisms provided by LLVM to make dependences from other passes explicit, influencing the scheduling of the pass during compilation. Furthermore, the compilation in *Sambamba* is separated into three phases, in each of which a *Sambamba* module can take part:

PRE_OPT is the initial phase in which a registered module gets to see the raw, unchanged input before any optimizations take place. Most of $ParA_\gamma$'s static parts operate in this phase. The representation of the application in this phase is very close to the original source and not "obfuscated" by aggressive optimizations. It is typically well suited for high level optimizations like automatic parallelization. Furthermore, $ParA_\gamma$ clones and stores an unoptimized version of each function

[1] `http://llvm.org/docs/WritingAnLLVMPass.html#the-modulepass-class`

containing parallelization candidates, which form the basis of runtime parallelization. Final optimization is left for the runtime parts as it has to *succeed* parallelization.

POST_OPT is the phase succeeding the execution of regular LLVM compiler optimizations, whose selection can be influenced via the typical command line options (e.g., *"-O1"* - *"-O3"*, or flags triggering individual optimizations).

In this phase $ParA_\gamma$ checks and reacts to severe changes performed by the potentially aggressive optimizations. One example is changes to the calling convention of functions, to which $ParA_\gamma$ reacts by adapting the previously cloned parallelization candidates accordingly.

LINK_TIME is the last phase during compilation right before linking everything together to form the final adaptive binary.

To transfer analysis results, like the parallelization candidates, for instance, from compile-time to runtime, *Sambamba* provides a so-called *DataStore* in which all serializable information can be stored. The content of this data store will be linked into the application binary and is available at runtime.

To hook into the runtime system, the dynamic component of a module registers an *init* as well as a *shutdown* method with the *Sambamba* runtime system. Not surprising, the *init* function is invoked to initialize the respective module *before* the actual application starts executing. During initialization, the module can read and manipulate all (command line) parameters given to the application. This is particularly useful to read and filter parameters targeted to the runtime modules themselves.

Typically, a *Sambamba* module reads the statically gathered analysis results from the *DataStore* provided by *Sambamba* and acts correspondingly. $ParA_\gamma$ for instance reads the parallelization candidates and, depending on the availability of profiling information gathered during earlier runs, either immediately installs parallel function versions, or profiling versions of parallelization candidates into the system. Afterwards it starts a thread running in parallel to the application reacting to significant changes in the available profiling information without influencing the application running in another thread.

Before the application terminates, *Sambamba* will invoke the *shutdown* method of each *Sambamba* module, giving it a chance to persist information for future executions.

8.1.3 Multiple LLVM Modules

One particular technical problem to which *Sambamba* offers a solution is the isolation of effects of multiple *Sambamba* modules running in parallel to each other and to the running application. Each module is free to create, change and install multiple versions of each function. *Sambamba* provides the facilities to easily do so.

One very important mechanism to prevent interference between multiple concurrently operating *Sambamba* modules is to keep multiple LLVM Modules. An LLVM Module, of which during a typical LLVM compilation only one exists, is a collection of global values and functions. It typically represents a whole application.

The *Sambamba* runtime in contrast keeps and synchronizes multiple interconnected LLVM Modules at the same time:

OrigMod is the original, statically compiled module of the application. It contains all functions in their statically compiled version, without any changes introduced by dynamic *Sambamba* modules. This module is read only and may not be changed.

Private Modules are kept for each *Sambamba* module to operate on. This way, the effects of the changes performed by each *Sambamba* module are isolated from the other *Sambamba* modules.

RunningMod is the module representing the currently running version of the application. "Installing a changed version of a function into the running system" actually means to transfer it from one of the private modules to the *RunningMod* and trigger just-in-time compilation. Registering and un-registering different versions $f_{P_1} ... f_{P_n}$ of a function f_O contained in the original application is the only way for a *Sambamba* module to change the *RunningMod*. Constructing these function versions has to be done in the private modules.

Sambamba orchestrates and synchronizes all accesses to the *RunningMod*. It takes care to copy over (or, more precisely, link) all symbols and globals accessed by the functions to be installed. One particularly important feature is that it is allowed for a call instruction $call_P$ ...(...) in a private module M_P to call a function f_O in *OrigMod*, a function f_P in M_P itself, or a function f_R in *RunningMod*. The consequences of choosing either variant are as follows:

call$_P$ f_O(...) calls the original, unmodified (and unmodifiable) function. In the parallelization context of $ParA_\gamma$ this means, for instance, to always call the sequential version of f. *Sambamba* will link a unique copy of f_O into *RunningMod* upon seeing the first reference to it.

This is the version used by the *tnd* dispatcher when switching to sequential execution as described in Subsection 7.4.2.

call$_P$ f_P(...) calls an individual copy f_P of f, privately owned by *Sambamba* module M_P. *Sambamba* will link f_P into *RunningMod* upon seeing the first reference to it. Multiple references to f_P will be linked to the same copy in *RunningMod*. Note that this way multiple different versions of a function f contained in the original application, but also completely new functions and globals, are installed into *RunningMod*.

call$_P$ f_R(...) calls the function version currently installed into the running system. Note that "currently" does not mean calling the function that is installed at the time of installing the calling function into *RunningMod*. It actually means calling the version of f that is installed at the time of invoking $call_P$ f_R(...). Installing new versions of f into *RunningMod* in the future will thus potentially influence the behavior of $call_P$ f_R(...).

This is the variant used by the *tif* dispatcher (see Subsection 7.4.3).

Note that in LLVM typically only one LLVM module exists at a time and it is illegal for a module to contain a reference into a different module. In *Sambamba* this is perfectly valid for the private modules only and a mechanism to guarantee isolation while minimizing the necessity for excessive code duplication and handling. *Sambamba* provides all means

to keep dealing with multiple modules as convenient and safe as possible. *RunningMod*, as well as *OrigMod* are supposed to be self-contained and thus not allowed to contain references to different modules. This invariant is protected by the *Sambamba* runtime system.

The typical flow of optimization performed by a *Sambamba* module using the different LLVM modules is to first request an initially empty private module M_P from the *Sambamba* runtime system. To optimize a function f it then creates a clone f_P of f_O taken from $OrigMod$. In this cloning process, it relinks all references to global values and functions of f_O from *OrigMod* to *RunningMod*. f_P is then changed in isolation as desired and finally installed into *RunningMod*.

8.2 $ParA_\gamma$—Relevant Implementation Details

Concerning the implementation of $ParA_\gamma$ the following techniques are of particular importance to understand the results achieved.

8.2.1 Block Splitting

While the principle ideas and techniques used by $ParA_\gamma$ are conceptually not limited to operating on a basic block level, in the current implementation we chose to do so for reasons of scalability. This is clearly a trade-off in the light of which the precision and power of parallelization might suffer, as the scheduler is limited in its freedom to independently schedule potentially costly instructions contained within the same basic block[2].

To counter this limitation in freedom, $ParA_\gamma$ seeks to isolate two kinds of instructions by extracting them from their containing block:

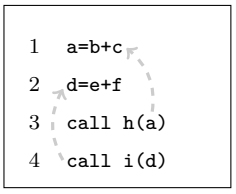
Potentially costly instructions are isolated to give the scheduler the freedom to schedule them in parallel to each other or the surrounding code. Costly instructions

[2]Theoretically the size of a basic block is not limited. The average number of instructions per basic block highly varies with the compiler and the source language for instance. Calder et al. [113] for instance give numbers of 5 to 8 instructions on average per basic block for C/C++ applications.

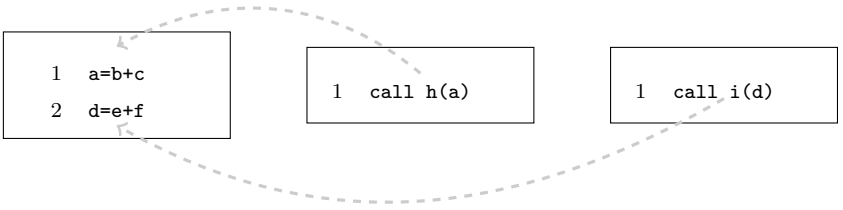
isolated by para include *call* and *invoke* instructions, except if the called function is known to be very short running.

Parallelization hindering instructions are isolated to break chains of dependences between basic blocks and give the scheduler the freedom to parallelize the code surrounding the isolated instructions. Examples of such instructions include the computation of loop iteration variables and reduction operations.

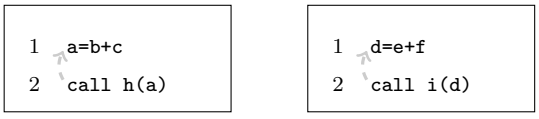
Extracting an instruction from a basic block is however not as simple as splitting the basic block before and after the respective instruction. Doing so typically introduces a dependence pattern that would equally likely hinder parallelization. Consider a basic block as follows:



Naively splitting this block to isolate both calls would result in three new blocks:



Unfortunately, both blocks containing the isolated calls do have a dependence to the first block containing the operand computation. The scheduler is now free to execute the calls in parallel to each other, but only *after* the argument computation has completed. This is an unnecessary restriction. The desired result instead is as follows:



This configuration does not impose any dependences between basic blocks and gives the scheduler all freedom to schedule the calls in parallel to each other.

Instead of performing naive splitting, $ParA_\gamma$ isolates complete intra-block dependence chains originating from a target instruction. In case multiple target instructions with non-disjoint dependence chains exist within the same basic block, dedicated basic blocks are created for the overlapping computation in order to maximize scheduling freedom within the bounds of preserving the sequential program semantics.

8.2.2 Schedule Cache

Upon a typical recompilation of an application after changing a source file, big parts of the code remain unchanged. $ParA_\gamma$ makes use of this fact to speed up the compilation process by re-using pre-computed parallel schedules (i.e., parallelization candidates). This way, it effectively saves the time to re-run the linear optimizer for each unchanged function.

The idea is motivated by *ccache* [114], which indexes a cache persisted on disk by computing a hash over the source to compile (after running the preprocessor). $ParA_\gamma$ follows this example but indexes its schedule cache by computing a deterministic hash over the PDG structure. This implies two things: the cache is operating on function granularity; and two functions having the same PDG share the same entry in the cache. As the PDG contains all memory effects, which is also reflected in the hash computation (in contrast to function or variable names and other identifiers, which are abstracted away), this is exactly the desired behavior.

In case $ParA_\gamma$ finds for a given PDG an entry in its schedule cache, the corresponding schedule is immediately taken and the ILP not even constructed.

8.2.3 ILP Cloud

In case $ParA_\gamma$ does not find a suitable pre-computed parallel schedule for a given PDG in its cache, it typically constructs multiple integer linear programs to find local parallelization candidates as described in thorough detail in Chapter 6. As solving those ILPs, some

of which are of non-negligible complexity, takes most of the compilation time, a second cache is introduced: the *ILP Cloud* is indexed by a hash over the ILP structure and stores feasible solutions for a given ILP.

After constructing the ILP for a given PDG block as described earlier, $ParA_\gamma$ checks the availability of a feasible solution to the ILP. In case an optimal solution exists, $ParA_\gamma$ takes it and does not need to run the optimizer. In case a *feasible* but non-optimal solution exists, $ParA_\gamma$ can, subject to the given configuration, decide to do two things:

1. Check if the CPU time spent computing this solution (which is also stored in the ILP cloud) is above its own ILP solving timeout, and if so, take the feasible solution as is.

2. If the CPU time spent so far to optimize the available solution is below $ParA_\gamma$'s own threshold, the feasible solution can be used as a starting point for the ILP solver, which in turn spends the difference of time to $ParA_\gamma$'s time budget in polishing the solution further towards an optimum.

3. $ParA_\gamma$ can, independently of the CPU time spent so far to solve the cloud solution, use its own time budget to polish the solution.

In any case, $ParA_\gamma$ will store a found and potentially improved solution in the ILP cloud for later reuse. In addition to instances of $Sambamba/ParA_\gamma$ which contribute to the solutions stored in the ILP cloud, we employed available compute resources to regularly take feasible but non-optimal solutions from the ILP cloud and further optimize them.

Note that as the ILPs constructed by $ParA_\gamma$ are based on local dependence DAGs which abstract away identifiers, control flow and program order and contain less than 10 nodes on average (see Table 9.1) chances are that ILP solutions can be shared between independent programs.

The implementation of the ILP cloud has been done together with Clemens Hammacher who implemented in particular the server side part of the cloud, including ILP serialization and deserialization.

8.3 A Note on Inter-core Communication

Recent successful work on parallelizing irregular applications [74, 81] has shown that, depending on the form of parallelism, inter-core communication latency is a limiting factor for successful parallelization. Helix [81, 82], for instance, is able to achieve impressive speed-ups on irregular applications without even relying on speculation. The approach however strongly relies on efficient inter-core communication, as loop-carried dependences have to be communicated between every pair of succeeding loop iterations which are executed on different cores in a round-robin fashion. By clever scheduling, the communication latency can be hidden to some degree but the scalability of the approach by design is limited by the communication latency.

The influence of inter-core communication on the successful parallelization using $ParA_\gamma$ is limited in several ways. Note that inter-core communication only happens when spawning a parallel task and only from the spawning to the spawned task. This communication is minimized in the ILP formulation by including the *ComCost* in the optimization function. Furthermore, task-blocking (see Section 7.3) greatly reduces the amount of necessary communication by executing multiple successive instances of a task, e.g., loop iterations, on the same core. Recomputable values, like for instance induction variables, are communicated only once per block instead of once per task instance. Values which are needed by multiple task instances, e.g., loop-invariant live-in values, are communicated once per block at most. For the remaining, strictly necessary communication, a work-stealing task scheduler[3] in the spirit of Cilk is employed by the $ParA_\gamma$ runtime to effectively make use of the cache hierarchy and minimize necessary inter-core communication, in particular for nested parallelism.

This chapter provided a technical system overview of *Sambamba* and gave insights into the implementation of *Sambamba* and $ParA_\gamma$. Block splitting is particularly important to maintain the necessary flexibility while scheduling on the basic block level for reasons of efficiency. A local schedule cache and a global ILP cloud mitigate

[3] The $ParA_\gamma$ runtime system relies on the dynamic scheduler of *Intel TBB*.

the problem of long running integer linear optimization at compile-time.

Chapter 9

Evaluation

In order to demonstrate the effectiveness and generality of $ParA_{\gamma}$ as described in the previous paragraphs we performed a detailed evaluation[1] on six selected benchmark programs originating from different benchmark suites. The programs have been selected because of their differing characteristics as shown in Table 9.1. For each of the selected programs, the table lists the following:

Suite, the source benchmark suite. The programs are taken from the *Barcelona OpenMP Task Suite* (BOTS) [115], the *Cilk* [41] suite of sample applications, as well as the *Parsec* [116], and the *PolyBench* [117] benchmark suites.

SLOC, the number of *source lines of code*, excluding empty lines and comments.

$N_{max/avg}$**,** the maximum and average number of equally control-dependent nodes, i.e., the children I_g of a *PDG* group node g as described in Subsection 6.1.2. This number, together with the following one, is the dominant factor influencing the complexity of the ILP to optimize during scheduling.

$E_{max/avg}$**,** the maximum and average number of dependence edges Δ_g in DAG_g, also as described in Subsection 6.1.2.

[1] This chapter is based on the evaluation conducted together with my co-authors Johannes Doerfert and Clemens Hammacher as part of the paper "Generalized Task Parallelism" [15].

TABLE 9.1: Characteristics of the programs used to evaluate $ParA_\gamma$. They cover a broad range of domains and parallelization schemes. Some of them require privatization or reduction recognition and handling.

					Enabling Techniques		Applied Scheme		
Benchmark	Suite	*SLOC*	$N_{max/avg}$	$E_{max/avg}$	Priv	Red	Loop (full)	Loop (partial)	Task
alignment	BOTS	612	93/9.86	128/9.27	✓	✓	✓		
cilksort*	Cilk	387	22/8.58	23/7.19				✓	✓
fft	Cilk	3168	63/8.92	161/10.93	✓	✓	✓		✓
blackscholes	Parsec	393	24/7.38	30/6.87			✓		
BiCG	PolyBench	1586	31/9.15	37/9.50		✓	✓		
gesummv	PolyBench	1582	24/7.88	31/8.08		✓	✓		

Enabling Techniques, the parallelism enabling techniques used to parallelize the application. Enabling techniques used in this evaluation are reduction (Subsection 5.2.1) and privatization (Subsection 5.2.2). We do not rely on speculation (5.2.3), which is not part of this thesis.

Applied Scheme, the parallelization scheme. We classify parallelization candidates into three (simplified) schemes as applied by existing parallelization approaches:

1. *Loop (full)* corresponds to *DOALL*-style loop parallelism: all iterations of a loop without any loop-carried dependences, apart from those induced by induction variable computation, can be executed in parallel to each other or in any arbitrary order.
2. *Loop (partial)* corresponds to loops with loop-carried dependences which do not prohibit to execute parts of the iterations in parallel to each other. Different parallelization approaches exist that deal with such loops. Examples include *DOACROSS*, *DSWP* [72–75, 77] or *Helix* [81, 82, 93].
3. *Task* represents parallelism independent of loops as can be expressed, for instance, in the *Cilk*, *Cilk++*, or *Intel Cilk Plus* languages. This form of

parallelism is also known as control parallelism. Special forms include fork-join parallelism.

Again, note that $ParA_\gamma$ does not *explicitly* exploit the mentioned forms of parallelism. Also, it does not implement or include a specialized approach for any of the mentioned parallelization schemes. Parallelization in $ParA_\gamma$ is solely based on dependences and does not take any particular program structure into account. The resulting parallel code produced by $ParA_\gamma$ nevertheless may be very similar to the result produced by more specialized approaches falling into the above mentioned categories. The purpose of this classification is to show that $ParA_\gamma$, by abstracting from the program structure, *implicitly* exploits such well known patterns of parallelism.

The six programs have been chosen as representatives for their particular style of parallelism as reflected in their originating benchmark suite. Later in this chapter we additionally show the results of $ParA_\gamma$ applied to all programs of the *PolyBench* benchmark suite as well as most[2] applications of the *Cilk* example suite.

For each of the detailed evaluation subjects we compare $ParA_\gamma$ against an approach which ships with or is usually evaluated on the respective benchmark suite. To evaluate the influence of the most important runtime techniques described in Chapter 7 four different configurations of $ParA_\gamma$ are used: with runtime dispatch (Section 7.4) enabled, with loop blocking (Section 7.3) enabled, both runtime techniques enabled and none of them enabled. All configurations, including the latter, make use of runtime profiling information (Section 7.1) to choose from different parallelization candidates (Section 7.2) and form one parallel version per function. Without such information, parallelization would be unguided.

In this evaluation, we particularly assess the following hypotheses:

1. $ParA_\gamma$ identifies and leverages different forms of parallelism;
2. $ParA_\gamma$ effectively makes use of privatization and reduction recognition; and
3. $ParA_\gamma$ creates efficient parallelized code over a broad range of applications.

[2] A few applications have been excluded due to technical restrictions as will be described later in this chapter.

9.1 Setup

All experiments were performed on an *Intel Core i7 920* quad core CPU with 2.67 GHz, 8 MB cache and *Hyper-threading*, allowing to execute 8 threads in parallel. The *LLVM* based approaches (*ParA*$_\gamma$ and *Polly*) as well as the sequential baseline were compiled using *clang 3.4.2*, the most recent version at the time of the evaluation; for other approaches, in particular for the *OpenMP* versions, *gcc 4.9.1 (pre-release)* has been used.

Note that, as mentioned earlier, *ParA*$_\gamma$ makes use of statically estimated as well as dynamically collected profiling information. It does so during static scheduling of parallelization candidates (see Subsection 6.1.1), as well as at runtime to select between different candidates or combinations thereof. For our evaluation, we did *not* take runtime profiling information into account during static scheduling, which is solely based on static estimates. The candidate selection at runtime however averages the profiles collected during the current and earlier runs of the binary and takes them into account. It is therefore able to use profiling information on actual inputs.

9.2 Benchmark Suites

PolyBench/C 3.2 by Pouchet [117] contains scientific codes dominated by loop execution. The programs have been selected due to their eligibility for polyhedral loop optimizations. We compare the performance of *ParA*$_\gamma$ against *Polly*, the polyhedral optimizer of the *LLVM* compiler framework. Note that the form of parallelism exploited by typical polyhedral optimizers is significantly different from the one exploited by *ParA*$_\gamma$: big improvements in terms of execution time are achieved by optimizing for cache locality, a goal that *ParA*$_\gamma$ does not explicitly share[3]. Therefore, we consider the benchmarks chosen from this domain particularly interesting as they show highly nested loops with very small bodies which do not justify the overhead of spawning parallel tasks for every instance. In order to efficiently parallelize these applications, *ParA*$_\gamma$ has to provide countermeasures to minimize the $\frac{overhead}{work}$-ratio.

[3] *ParA*$_\gamma$ does take cache locality into account, for instance when allocating privatized storage for reduction, privatization and the collection of execution profiles. It is not the primary optimization goal though.

The programs taken from the *Cilk* [41] example suite are compared against optimized execution in *Cilk*. For the sequential reference, we compiled the *serial elision*, which is the result of deleting all *Cilk*-specific language constructs (*spawn*, *sync*, ...) from a Cilk program, using the *clang* compiler, in order to avoid any overhead induced by the *Cilk* runtime system. The *BOTS* suite of Duran et al. [115] contains one sequential and multiple manually parallelized program versions annotated using *OpenMP* pragmas. In all our experiments, we compare against the best performing individual *OpenMP* variant. The *Parsec* suite [116] contains hand-crafted versions of each program, parallelized using *OpenMP*, *Intel TBB* and *native POSIX threads*. Again, we compare against the best of those versions.

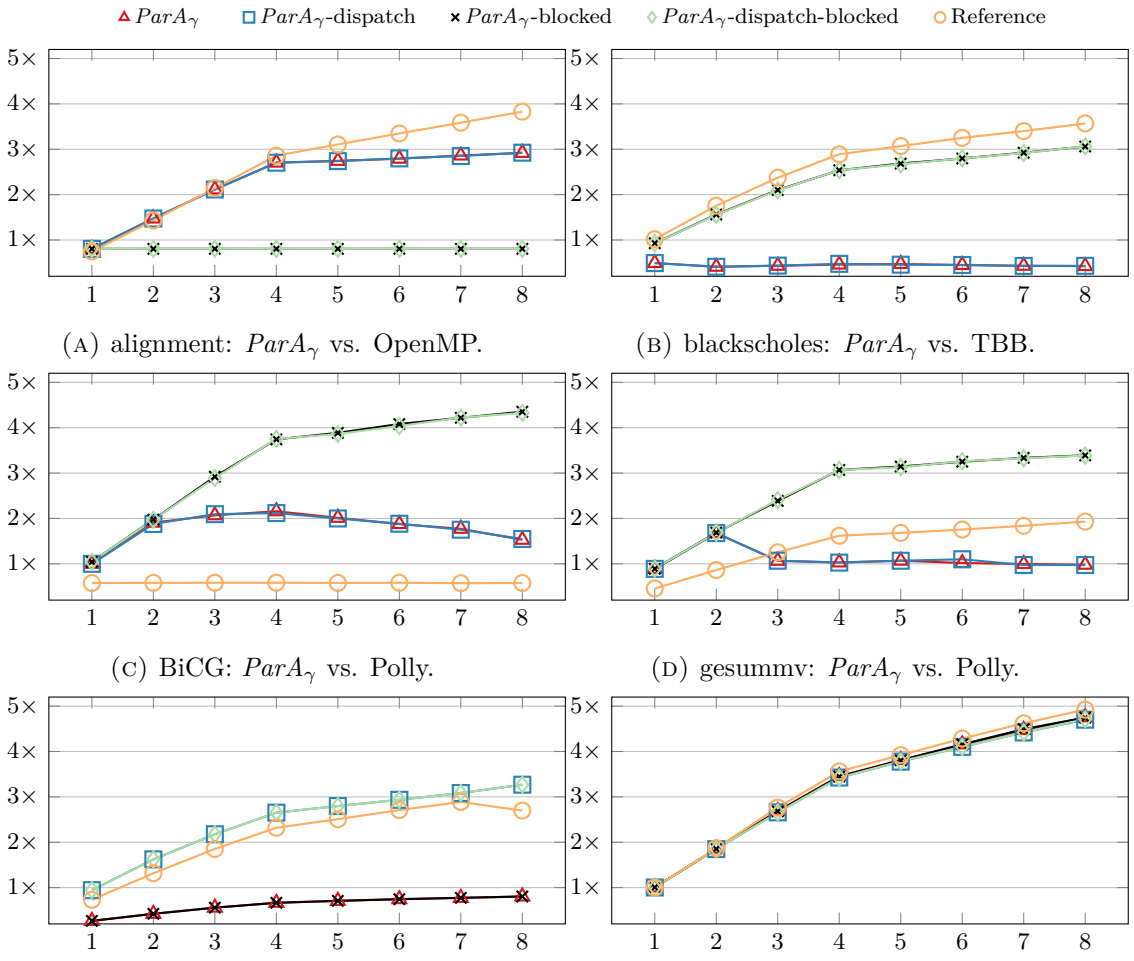
9.3 Results of the Detailed Evaluation

Figure 9.1 shows the result of the detailed evaluation. All numbers are normalized against an optimized sequential program version compiled using *clang*.

In the evaluated programs, $ParA_\gamma$ is able to detect all program locations that were also parallelized by the domain experts. In all cases $ParA_\gamma$ is able to significantly decrease the execution time as compared to sequential execution. We validated the statistical significance of all reported speedups with a *confidence of 99 percent* using the *Speedup-Test* described by Touati et al. [118]. Our results show both the *generality* and the *effectiveness* of our $ParA_\gamma$.

Detailed Explanation

In the *alignment* (9.1a) and *blackscholes* (9.1b) programs, the efficiency of the $ParA_\gamma$-parallelized versions falls behind that of the hand-crafted versions using *OpenMP* or *TBB*. The difference stems from two factors: First, the *OpenMP* and *TBB* programs are compiled with gcc, which in these particular cases is able to create more efficient code. Second, the $ParA_\gamma$ runtime system introduces overhead for allocating heap space for the parallel tasks and for privatization and reduction locations. This overhead is significantly higher than that of the reference systems.



(A) alignment: $ParA_\gamma$ vs. OpenMP.

(B) blackscholes: $ParA_\gamma$ vs. TBB.

(C) BiCG: $ParA_\gamma$ vs. Polly.

(D) gesummv: $ParA_\gamma$ vs. Polly.

(E) cilksort*: $ParA_\gamma$ vs. Cilk.

(F) fft: $ParA_\gamma$ vs. Cilk.

FIGURE 9.1: Evaluation of $ParA_\gamma$ on six programs from different domains, containing different styles of parallelism. The x-axis shows the number of threads; the y-axis compares *speedup over sequential execution*. Parallelization in *OpenMP*, *Cilk*, and *Intel TBB* is done manually by experts; *Polly* and $ParA_\gamma$ parallelize automatically.

We see that *alignment* profits from neither of blocking or runtime dispatch. Indeed, blocking harms performance. This is because the parallelized loop does not have enough iterations to reach the chosen block size. As a result, blocking effectively sequentializes the execution. If the choice to enable blocking is left to the $ParA_\gamma$ runtime system, it therefore disables it based on collected runtime profiling information showing that the iteration count of the loop is too low for blocking to be possibly profitable. The performance improvements achieved by $ParA_\gamma$ on the *blackscholes* benchmark in contrast heavily benefits from blocking.

The *alignment* program is particularly interesting, as parallelizing the main loop in the

pairalign function requires the privatization of no less than 15 variables at different nesting levels of the loop nest. This necessity is reflected in the *OpenMP* annotations of the handcrafted parallel version[4]; if one of those annotations is missed by the developer, the executable will produce incorrect results. $ParA_\gamma$ is able to automatically find variables to privatize and produces the corresponding code to guarantee correctness without human guidance. In its semi-automatic mode of operation, $ParA_\gamma$ will present all necessary privatization locations to the programmer as described in Chapter 10 and offers to automatically take care for privatization.

In the *BiCG* (9.1c) and *gesummv* (9.1d) programs from the PolyBench suite, $ParA_\gamma$ outperforms *Polly*, the specialized tool for the kind of programs contained in this suite. We can see that blocking is the enabling measure of $ParA_\gamma$'s performance benefits, whereas dynamic dispatching does not contribute. It does, however, also not impose any significant observable overhead. Due to the loop dominated execution of both programs this is not surprising.

It is, however, surprising that *Polly* is seemingly even harming the performance of the applications. In case of *BiCG* it has mainly two problems:

- *Polly* works on basic block level and is unable to split blocks on demand. Both statements of the innermost loop (see Figure 1.4a) share the same basic block which consequently induces loop-carried dependences over both containing loops.
- *Polly* is unable to deal with reductions and therefore misses an important opportunity of parallelization.

The slowdown of *Polly* in this particular benchmark comes from the fact that it not only misses the profitable parallelism, but also parallelizes the loop which initializes the *s* array to 0, which is not profitable.

In *gesummv*, *Polly* finds the right location to parallelize but the generated code based on *OpenMP* is unable to profitably exploit the exposed parallelism. The speedup would improve, if *Polly* was able to additionally vectorize the generated code, which it is not in this case.

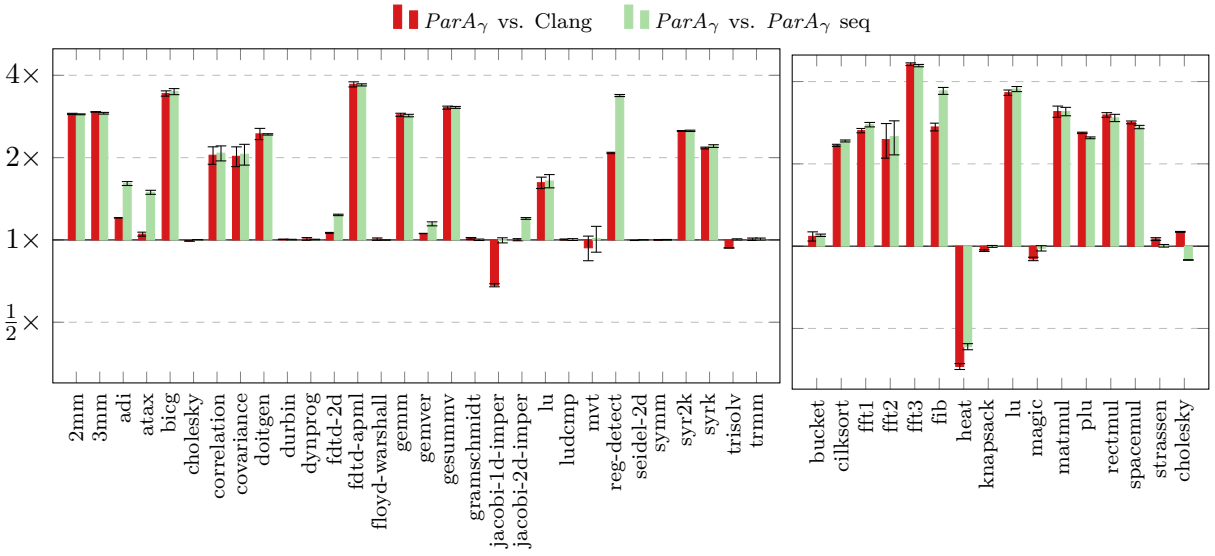
[4] Appendix B shows the *OpenMP*-parallelized version taken from the *BOTS* suite

Note that after the findings and insights resulting from this particular evaluation, *Polly* has been improved to at least partially overcome the above mentioned limitations. In particular capabilities to detect and exploit reductions have been extended as described in our own work [16].

The *fft* (9.1f) and *cilksort** (9.1e) programs implement *Fast Fourier Transform* and a standard *mergesort*. Note that the version of cilksort as it is contained in the Cilk suite performs a switch from *mergesort* to *quicksort* at a hard coded array size boundary. Cilk and $ParA_\gamma$ (without runtime dispatch) both profit from this boundary when automatically parallelizing as it effectively causes execution to switch from parallel to sequential once the problem size falls below a given size. As for regular targets of a parallelizer such help cannot be expected, we removed this boundary for our benchmarks (hence the * in *cilksort**). As demonstrated in Subsection 7.4.2 (see Figure 7.5) $ParA_\gamma$ makes placing such somewhat artificial boundaries superfluous.

As mentioned earlier, $ParA_\gamma$ is able to make use of dependence annotations placed in the code by the programmer. Like essentially all applications of the Cilk example suite *fft* and *cilksort* mainly consist of recursive functions. As described in Subsection 5.1.1, $ParA_\gamma$ profits from user provided annotations in such cases and we manually annotated relevant parts. The idea is similar to that of Vandierendonck et al. [79]: hints are only used to improve dependence information while both parallelization and parallel execution stay fully automatic.

As we can see from the results neither of *cilksort** and *fft* profits from blocking as the dominating parallelism does not stem from parallel loops. *fft* also does not profit from runtime dispatch. This is mainly due to a highly specialized and hand-optimized implementation which switches over to specialized function versions to solve smaller sub-problems: specialized implementations which are not parallelized. This corresponds, just like in the original version of *cilksort*, to an implicit switch from parallel to sequential execution. Later on in this chapter, we will give further insights into the specialized and input dependent execution paths through the *fft* application.



(A) PolyBench. (B) Cilk suite.

FIGURE 9.2: Speedups achieved by $ParA_\gamma$ on the PolyBench 3.2 9.2a and Cilk suite 9.2b with 8 threads on a quad core with Hyper-threading.

*cilksort** however greatly benefits from runtime dispatching. Indeed, only with runtime dispatching $ParA_\gamma$ is able to achieve any speed-up. In that case, however, it constantly outperforms the Cilk version.

Note that a significant slowdown can be observed for $ParA_\gamma$ without dispatching, even when using only one thread. This is because the overhead of creating and scheduling parallel tasks is introduced at every level of recursion and for every problem size. As discussed in detail in 7.4.2 this overhead outweighs the actual productive work, in particular towards the leaves of the recursion tree. Additionally, task stealing hurts data locality, leading to the observed slow-downs for multi-threaded execution.

For both Cilk applications $ParA_\gamma$ with runtime dispatch is able to fully match the parallelization decisions of the manually crafted implementations; performance with runtime dispatching enabled is comparable to the performance of the Cilk version of both benchmarks.

9.4 Results on *PolyBench* and the *Cilk suite*

In addition to the detailed benchmark evaluation shown in Figure 9.1 we evaluated $ParA_{\gamma}$ on all applications of the *PolyBench/C 3.2* Suite [117], as well as most of the applications of the suite of Cilk [41] example applications. The results are shown in Figure 9.2a and Figure 9.2b[5] respectively.

All experiments have been conducted on the same machine as the previous experiments. Speed-ups are relative to an optimized binary produced by clang and sequential execution in the $ParA_{\gamma}$ framework respectively. The purpose of the latter is to demonstrate the speed-up obtained by parallelization alone. It ignores the overhead introduced by the *Sambamba* runtime system, for instance just-in-time compiled execution. Another reason for differences in the numbers is the potential lack of inter-procedural program optimization of the parallelized functions. These are not performed in order to be free to exchange any function at runtime. Function inlining, a prominent inter-procedural optimization, for instance, would prohibit to flexibly exchange and execute the inlined function independently at runtime.

In the case of PolyBench, the reported speedups are obtained using the own timing measurement facilities of the benchmark suite, and using the large input set. The speedup refers to the main computational kernel of each benchmark and not the whole application.

As explained earlier, for the Cilk applications the serial elision is computed and annotated with dependence hints prior to automatic parallelization. We had to exclude four applications (*ck*, *game*, *queens* and *kalah*) from our evaluation as the serial elision of those programs was not easily computable due to the use of Cilk inlets. Furthermore, we excluded *hello*, which is a trivial hello world program, as well as *nfib*, which is basically the same as *fib*. For each of the remaining applications we measured the overall program speedup as not all applications come with their own measurement of relevant program parts. While it seems to be the most appropriate way to us to treat all benchmarks of the suite in the same way, measuring whole program speedup results in lower speed-ups than one might wish to see on a quad-core machine with Hyper-threading. In most cases

[5] Earlier results similar to those shown in Figure 9.2b have been reported in our own earlier work [17].

this is caused by large parts of the application being inherently sequential: for instance, allocating and filling large arrays to sort and finally verifying result correctness. For *cilksort*, *matmul*, and *spacemul* the difference is significant: While their measured kernels have been accelerated by factors of *4.32*, *3.63* and *3.26* respectively, the overall speed-up presented in this evaluation is significantly lower.

For *fft* we report three different performance numbers. They represent the same program run on different inputs, each covering a different characteristical execution path through the application:

fft -n 10,000,000: *fft1* is executed using the argument *10,000,000*, which triggers the general path through the *fft* benchmark and computes the fast fourier transformation on an array of 10,000,000 components.

fft -n 33,554,432: *fft2* triggers a heavily optimized path through the benchmark which is tailored to powers of two as input size ($33,554,432 = 2^{25}$). In this case, the performance improvement achieved by $Sambamba/ParA_{\gamma}$ is not quite as high as for the general case.

fft -c: *fft3* triggers a special checking mode of the benchmark which executes the *fft* computation in a loop and checks correctness of the results. In addition to parallelizing the *fft* computation itself, *para* is able to parallelize this outer loop, which results in the highest speedup achieved by $ParA_{\gamma}$ among the three different *fft* inputs.

This particular *fft* implementation is an example of the input dependent achievable parallelization speedup. Note though that not the parallelization of any given function depends on the input, but instead the chosen path of execution (i.e., the executed functions), and with it the overall speedup of the application. Such behavior in general poses a challenge to parallelization approaches depending on previously collected profiling information as the expected benefit heavily depends on the developer to choose the right representative inputs to collect profile information. $ParA_{\gamma}$ in contrast is independent of such pre-selection of inputs. Its runtime system is able to dynamically react to previously unseen execution paths being taken.

The results of the evaluation presented in this chapter confirm our hypotheses:

1. $ParA_{\gamma}$ subsumes different parallelization approaches by effectively detecting and leveraging different forms of parallelism.
2. Parallelization enabling techniques like privatization and reduction recognition are used where applicable.
3. The runtime is comparable to state-of-the-art parallelization tools, but no developer guidance is needed.

Chapter 10

Extension and Use Case: Semi-Automatic Parallelization

Fully automatic parallelization has been the goal of research for multiple decades and successful approaches arose, at least for certain domains. Nevertheless, and despite the fact that manual parallelization is considered hard and error-prone by the majority of programmers (see, e.g., Christmann et al. [4]), automatic parallelization still did not find its way into main-stream compilers, and with it into the domain of general purpose applications. Reasons, among others, are:

1. Limited capabilities of static analyses, for instance to precisely compute, or at least predict parallelization-limiting dependences (cf. Niall et al. [11]).
2. Limited and input-dependent predictability of expected performance.
3. Limited trust of developers in the capabilities and correctness of automatically parallelizing compilers.
4. Hard debugging of produced binaries involving a complex runtime system.
5. Limited trust of developers in their own comprehension of multicore and manycore programming [4] and execution environments combined with limited tool support.

Libraries and frameworks like *OpenMP* [7], or *Intel TBB* [40], domain-specific languages, and language extensions like *Cilk* [41], or *Intel Cilkplus* [42], and modern languages with parallelism included like *Scala* [46, 47], *Go* [45, 119], or *Rust* [120, 121] ease some of the problems and gain thrust. Their parallelization capabilities are, however, still mostly used and understood by experts only. What those tools have in common, however, is the fact that they expose parts of parallelization, including the responsibility for correctness and performance implications, to the programmer while reducing boiler-plate work and the risk of failure.

OpenMP is a particularly well adopted library for parallelization of C/C++ and FORTRAN applications and has been widely accepted by programmers as low level enough to leave the impression of being in control, and high level enough to significantly reduce the necessary boiler-plate work involved in parallelization. However, it still leaves the programmer alone with the responsibility of correctness and performance improvements. Appendix B shows the code of the *pairalign* function taken from the *alignment* benchmark of the *Barcelona OpenMP Task Suite* (BOTS) [115]. As also explained in Chapter 9, this code requires the programmer to cause the privatization of *15* variables to guarantee the correctness of the parallelized code. Only one missed location will leave the programmer with debugging a non-deterministically behaving parallel application.

Motivated by the acceptance and ease of use of *OpenMP* and to provide the possibility to easily guide or focus some of the costly analyses performed by $ParA_{\gamma}$, we have extended $Sambamba/ParA_{\gamma}$ and embedded it in a workflow of semi-automatic parallelization. An overview of the extended workflow can be seen in Figure 10.1.

Key difference to the regular flow of parallelization described in the earlier chapters is the possibility of the programmer to interact with the parallelization process by introducing *OpenMP*-like parallelization hints into the code and reacting to hints, warnings and errors reported by $ParA_{\gamma}$. The main difference to manual parallelization using *OpenMP* is the added programmer confidence by guaranteeing correctness of the parallelization, and automization of standard but error-prone parallelization-related tasks like privatization and reduction. Furthermore, the programmer profits from parallelization-related optimization

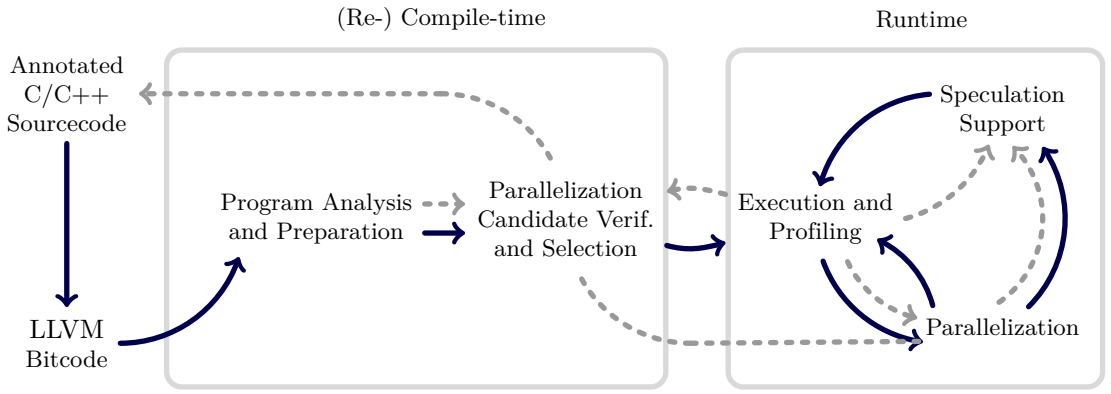


FIGURE 10.1: Overview of the $ParA_\gamma$ toolchain of semi-automatic parallelization. Solid arrows depict the flow of application code, dashed arrows the flow of analysis information.

of the code not usually done by regular compilers. The three key enabling components of this workflow are described in the remainder of this section.

- C/C++ language extension in the form of *pragmas*, resembling well-known and accepted *OpenMP* directives,
- the capability of $ParA_\gamma$ to communicate analysis results in the programmers terminology, and
- integration into the IDE to enable easy comprehension of the analysis results and hints.

10.1 C/C++ Language Extension

Exploiting the acceptance and familiarity of *OpenMP*, we have extended the *Clang* C/C++ compiler frontend of the *LLVM* ecosystem to process three intuitive and easily comprehensible parallelization directives:

#pragma sambamba parallel section {...} delineates a section of parallel execution. Conceptually this directly maps to a parallel section in the *ParCFG*. Without further directives, this section will however contain only one task. It will thus not introduce any parallelism.

#pragma sambamba parallel loop ... marks a parallel loop. It precedes a regular loop construct and consequently comes in three main flavors: as *for*, *while*, or *do ... while* loop. Similar to the parallel section directive, this directive does not itself introduce parallelism, which requires at least one task statement as follows:

#pragma sambamba parallel task {...} marks a task to be spawned off for execution in parallel to the code surrounding it within the same parallel section or loop, in which a parallel task always has to be contained.

Code surrounding marked parallel tasks in the parallel section containing it is put into an additional, implicit task; code surrounding the tasks of a parallel loop is put into the one reentrant task (see Subsection 4.1.3) of the parallel loop.

In contrast to corresponding *OpenMP* directives, these *Sambamba* directives do not simply and unconditionally introduce parallelism at the defined locations. Although the relation to the *ParCFG* as the result of automatic parallelization in *Sambamba*/$ParA_\gamma$ is obvious, it is a long way from the frontend to the *ParCFG* generated at runtime.

All desired parallelism exposed using the directives is first verified. Only in case the static parallelization analysis is able to prove correctness and the absence of possible race conditions, parallel schedules are generated matching the parallelization decisions of the programmer. In case static analyses determine possible conflicts, $ParA_\gamma$ offers to automatically fix them by introducing reduction or privatization code, the standard parallelism enabling tools, in case this is possible, i.e., the respective preconditions are fulfilled for the conflict inducing memory locations.

If reduction and privatization are no option to fix potential conflicts, and again assuming all necessary requirements like the absence of circular task dependences are fulfilled, $ParA_\gamma$ offers to guard parallel execution using its potentially very costly speculation mechanisms.

As a last resort in case none of the countermeasures is applicable, $ParA_\gamma$ points at the potential conflict inducing memory locations and marked tasks and asks the programmer to take responsibility for parallelization. The programmer can then force $ParA_\gamma$ to parallelize the code in case the warning is the result of the limited capabilities of $ParA_\gamma$'s static analyses or the race condition is desired. To force parallel execution without a guarding

speculation system and take over the responsibility, $ParA_\gamma$ offers two additional versions of the parallel section and loop directives respectively: "#pragma sambamba parallel section_nospec {...}" and "#pragma sambamba parallel loop_nospec {...}".

Finally, the programmer has the option to specify to the compiler how the parallelism annotations should be treated: the only and maximal parallelism in the compiled application, or the minimal parallelism to introduce. In the former case, $ParA_\gamma$ tries to automatically enable parallel execution of the marked program parts. In the latter case $ParA_\gamma$ tries to extend parallel execution and find more parallelism in the application than is marked by the programmer. In any case the resulting parallel loops and sections will profit from the whole range of $ParA_\gamma$'s runtime capabilities as described in Chapter 7.

The *Sambamba* directives should be understood as a way to give the programmer the desired degree of control over the parallelization process and to provide domain-specific knowledge about where parallelism is expected by the programmer to $ParA_\gamma$. The decision on how the parallelism should be best implemented and if it can possibly be profitably implemented should be left to the compiler and an accompanying runtime system as described in this thesis.

10.2 Communicating Analysis Results

A minimum requirement to increase acceptance and to open parallelization to a broader range of non-expert programmers is the clear and easily comprehensible communication of parallelization related errors, warnings and questions to the programmer. In semi-automatic operation $ParA_\gamma$ therefore carries information similar to debug information through the whole parallelization process. This information in particular contains the names of relevant variables and line numbers of relevant code regions and parallel tasks.

A difficulty is the fuzziness introduced by the abstraction of the *DS-Analysis* (see Subsection 4.1.1), which potentially unifies the information about memory locations, which $ParA_\gamma$ consequently cannot distinguish any more during later conflict analyses. In case of a detected conflict, $ParA_\gamma$ might thus present multiple possibly conflict inducing variables. The number of candidates can however be reduced by analyzing the set of variables touched

in the conflict inducing tasks, which in many cases leaves a single candidate to be reported to the programmer.

10.3 IDE Integration

To make the semi-automatic parallelization toolchain presented so far easily accessible, it has been integrated as a plugin into the widespread *Eclipse IDE*. Figure 10.2 shows a screenshot of the main features: two source editors are open showing different parallel constructs; parallelization related messages and warnings are highlighted in-place and also presented in the list at the bottom of the screen. A tool-tip gives detailed information on conflict-inducing variables.

The left excerpt from the code of a simple raytracer contains a parallel loop, which has been successfully verified by $ParA_\gamma$. It is highlighted in yellow since multiple memory locations need to be privatized to guarantee correctness. $ParA_\gamma$ indicates the entailed cost. Tasks not requiring any parallelism enabling techniques would be highlighted in green.

The right code editor in the screenshot shows an excerpt of the *cilkmerge* function taken from the *cilksort* program of the *cilk* suite of applications. $ParA_\gamma$ is unable to prove correctness and offers to enable the speculation system at runtime. The programmer facing this warning can take a deeper look at the reported conflict inducing variables and eventually provide the guarantee that no conflicts are possible.

This chapter presented a way to interactively use the automatic parallelization capabilities of *Sambamba*/$ParA_\gamma$. By using the presented C++ language constructs, the programmer is able to provide domain specific knowledge to the automatic parallelizer. In exchange $ParA_\gamma$ will free the developer from having to write error-prone and hard to maintain boiler plate parallelization code and provides safety and confidence by guaranteeing correctness of parallel execution.

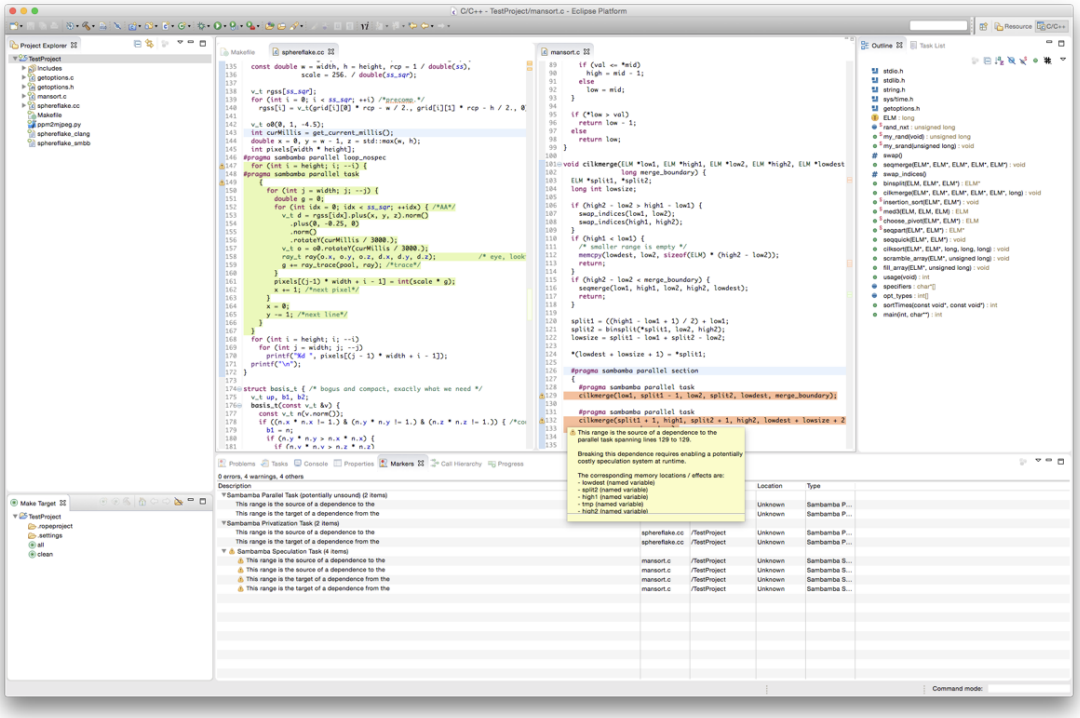


FIGURE 10.2: Screenshot of the *Sambamba/ParA*$_\gamma$ IDE integration showing multiple features: C/C++ editors with parallel task highlighting showing the result of parallelization analysis; error markers, warnings and tool-tips pointing at potentially problematic memory locations.

Chapter 11

Conclusion and Future Work

Parallelism, explicit or implicit, is here to stay facing the ubiquity of multi-core and many-core systems in combination with the stagnating or even decreasing single-core performance nowadays. Nevertheless, no practical solution has been found yet to support developers in the tedious and error-prone task of writing and maintaining their applications for such systems.

In this thesis, we presented $ParA_{\gamma}$, an approach to naturally unify different forms of loop parallelization as well as fork-join-style task parallelization, reduction, privatization, and speculation. We express the freedom to choose from all these alternatives in an integer linear programming approach to PDG scheduling that considers all parallelization opportunities at once to statically identify for each function a set of local parallelization candidates, whose instantiation is left to a flexible runtime system selecting the best combination with respect to a given cost function.

Facing the diversity and complexity of modern processors, memory systems, runtime environments, and application inputs, no static approach will ever be able to predict the profitability of a particular parallel code version. Therefore, the described approach relies on an adaptive runtime system to continuously recombine and reassess parallelization decisions and to adapt to changing requirements.

We furthermore presented an integration of the automatic parallelization capabilities of $ParA_\gamma$ into a modern integrated development environment, allowing the programmer to comprehend the necessary steps of parallelization and to provide important domain-specific knowledge to the compiler. Domain-specific knowledge, that no static analysis will ever be able to derive automatically. The semi-automatic parallelization capabilities of $ParA_\gamma$ effectively reduce the necessity to write boiler-plate parallelization code and help to minimize the risk of race conditions caused by dependences missed during manual parallelization.

We validated experimentally that $ParA_\gamma$ detects and effectively exploits parallelism in a variety of programs from many different benchmark suites exhibiting different kinds of parallelism. No single given parallelization approach known to us is able to exploit the same range of different forms of parallelism that $ParA_\gamma$ does. $ParA_\gamma$ with its adaptive runtime system is able to consistently achieve speedups at the same level or better than state-of-the-art parallelizing tools, or manual parallelization.

While we are convinced that the described approach is an important step towards unifying several important parallelization approaches, much is left to be done: our particular prototypical implementation of *Generalized Task Parallelism* has several limitations, mostly of technical nature. Despite not being a fundamental flaw of the approach, these limitations keep our implementation from parallelizing code which it could handle in principle.

In the following we give a short and non-exhaustive list of some relevant technical limitations and ideas on possibilities to address them in the future. The purpose is to make such limitations explicit and to foster future improvements of our implementation. They should not be understood as limitations of *Generalized Task Parallelism* as such.

Additionally, we mention two particularly interesting future research directions to extend and improve $Sambamba/ParA_\gamma$, which we always wanted to address but unfortunately did not yet find the time to do so.

Flow-insensitivity The fact that the data-structure analysis we use [23] is flow-insensitive causes imprecision when trying to identify memory regions as disjoint. This behavior

can of course be avoided by employing a flow-sensitive analysis. The problem is that flow- and context-sensitive analyses usually do not scale very well. This issue could be addressed by using a staged approach of dependence analysis as proposed by Hardekopf and Lin [122], or a client-driven one as for example the one of Guyer and Lin [123].

Dependence analysis With the goal of parallelizing general-purpose applications we chose to use DSA, which is a points-to analysis that is particularly well suited for irregular data-structures. However, this analysis has two major weak-spots relevant in our situation: it is not able to precisely deal with regular data-structures like arrays, and it greatly over-approximates the effects of recursive functions.

A student of ours has been working on a new approach combining and extending ideas of the range analysis by Rugina and Rinard [56], and the runtime parametric memory access analysis of Rus et al. [124]. The approach has not been fully integrated into $ParA_\gamma$ yet.

Feedback by the Speculation System As described earlier in this thesis, *Sambamba* in its current implementation provides different speculation systems. The implementation of $KTLS^+$, the most promising of these systems, was however completed only recently and has thus not been fully integrated into $ParA_\gamma$. Apart from only *using* a speculation system, which $ParA_\gamma$ can do, the static estimation of the speculation penalty as described in Sections 5.2.3 and 6.1.2 needs to be replaced by real dynamic feedback from the speculation system. Even more importantly, this also needs to be reflected in the cost function of the parallelization candidate composition of the $ParA_\gamma$'s runtime system as described in Section 7.2.

Apart from simply deciding for or against speculation or any given speculation system, it would also be very interesting to select and tune parameters of such a system as has been frequently described in the relevant research work. Reflecting such parameters in the process of scheduling and making them an integral part of the optimization space might be an interesting future research direction.

Parameter Dependent Dynamic Dispatch The different dynamic dispatch mechanisms described in Section 7.4 are able to effectively prevent from the oversubscription

of the executing platform and supersede and even outperform the parallelization boundaries regularly introduced during manual parallelization. They decide, based on explicit properties of the execution environment (number of available compute cores, system load, number of tasks in flight) and implicit properties of the parallelized application (task nesting depth), when to dynamically switch between the sequential and a continuously adapted parallel version of a called function. This is in contrast to the said manually introduced parallelization boundaries, which usually decide based on properties of the input, like for instance the size of the array to sort.

We have shown in Section 7.4 that relying solely on input properties and ignoring the availability of idle compute resources is not the best option. Completely ignoring input properties, however, neither is.

We therefore propose to extend $ParA_\gamma$'s dynamic dispatch capabilities by introducing a new dispatcher being able to derive relevant input properties on which to base the decision to proceed sequentially or in parallel. In a second step, this could be extended by making use of the input properties checked before entering a function to specialize the called parallel version. This implies the possibility to of more than one parallel version of any given function to exist at a time among which the input dependent dispatcher is able to dynamically select.

Sambamba, and independently $ParA_\gamma$, have been designed and implemented with reusability in mind. And while the quality of the implementation, in particular of those parts whose development has been driven by deadlines, could certainly be improved, we are convinced that the concepts, ideas and also big parts of the implementation are a worthy contribution to the field of parallelization research. We would be happy to see that this particular thesis helps to show that effectively and efficiently exploiting different shapes of parallelism using a single, unifying approach is possible. We are convinced that this also holds true for applications using irregular data structures and memory access patterns, sometimes referred to as *general purpose applications*.

List of Figures

List of Tables

Appendix A

Irregular Sample Application Written in C

```
1  /* Linked List Hash (ll_hash)
2   *
3   * ll_hash is a demo application that demonstrates some of the parallelization capabilities of
4   * the sambamba framework for runtime adaptive parallelization (http://www.sambamba.org).
5   *
6   * It constructs two singly linked lists of lengths chosen by the user. Each element of the
7   * list contains an integer, also chosen by the user (each element contains the same value).
8   * A hash value is then computed for the whole list by combining the hash values of the nodes.
9   *
10  * The integer value in the nodes controls the runtime of the hash computation for a single node,
11  * which grows exponentially in the given value. */
12
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <sys/time.h>
16 #include <math.h>
17
18 typedef struct list {
19     struct list *next;
20     int data;
21 } list;
22
23 list* makeList(int elemSize, int num) {
24     list *newNode = malloc(sizeof(list));
25     newNode->next = num ? makeList(elemSize, num-1) : 0;
```

```
26      newNode->data = elemSize;
27      return newNode;
28  }
29
30  void freeList(list *x) {
31      if (!x) return;
32      x->data = 0;
33      list *tmp = x->next;
34      free(x);
35      freeList(tmp);
36  }
37
38  long hashElem(list *elem) {
39      long n = (1 << elem->data);
40      long res = 0;
41      while (--n) res = 31 * res + n;
42      return res;
43  }
44
45  long hashList(list *x) {
46      if (!x) return 0;
47      return hashElem(x) + 31 * hashList(x->next);
48  }
49
50  long performTask(int elemSize, int listSize) {
51      list *x = makeList(elemSize, listSize);
52      list *y = makeList(elemSize, listSize);
53
54      long hashX = hashList(x);
55      long hashY = hashList(y);
56
57      freeList(x);
58      freeList(y);
59
60      return hashX * hashY;
61  }
62
63  struct timeval start, end;
64  int main(int argc, const char **argv) {
65      unsigned int iterations, wpn, size;
66      char lineBuf[128];
67
68      printf("Please specify: <iterations> <work per node> <length of list>\n");
69      do {
70          printf(" -> ");
```

```
71          if (!fgets(lineBuf, 128, stdin))
72              if (feof(stdin))
73                  break;
74              else
75                  continue;
76          int read = sscanf(lineBuf, "%u %u %u", &iterations, &wpn, &size);
77          if (read < 1) {
78              printf("!!! You have to specify an iteration count...\n");
79              continue;
80          }
81          if (read < 2) wpn = 21;
82          if (read < 3) size = 10;
83          double secSum = 0.0;
84
85          unsigned int i;
86          for (i = 1; i <= iterations; ++i) {
87              gettimeofday(&start, 0);
88              long result = performTask(wpn, 1 << size);
89              gettimeofday(&end, 0);
90              double secs = (end.tv_sec - start.tv_sec) + 1e-6 * (end.tv_usec - start.tv_usec);
91              secSum += secs;
92              printf("    %2u: result: %ld, took %7.3f s\n", i, result, secs);
93          }
94
95          printf("    ------------------------------------------------\n");
96          double avgSecSum = secSum / iterations;
97          printf("    Average of %u iterations: %7.3f s\n", iterations, avgSecSum);
98          printf("    ------------------------------------------------\n");
99      } while (1);
100     return 0;
101 }
```

Appendix B

OpenMP-parallelized *pairalign* Function

The following manually *OpenMP*-parallelized version of the *pairalign* function is taken as is from the *alignment* benchmark. It shows the parallelization related complexity of code that is still unavoidable despite the fact that *OpenMP* saves the programmer from a lot of the parallelization related boiler-plate. Please note the necessary privatization of 15 variables (lines 17 and 31) to guarantee the correctness of the parallelized code.

```
1  int pairalign()
2  {
3     int i, n, m, si, sj;
4     int len1, len2, maxres;
5     double gg, mm_score;
6     int    *mat_xref, *matptr;
7
8     matptr   = gon250mt;
9     mat_xref = def_aa_xref;
10    maxres = get_matrix(matptr, mat_xref, 10);
11    if (maxres == 0) return(-1);
12
13    bots_message("Start aligning ");
14
15    #pragma omp parallel
16    {
```

```
17    #pragma omp single private(i,n,si,sj,len1,m)
18       for (si = 0; si < nseqs; si++) {
19          n = seqlen_array[si+1];
20          for (i = 1, len1 = 0; i <= n; i++) {
21             char c = seq_array[si+1][i];
22             if ((c != gap_pos1) && (c != gap_pos2)) len1++;
23          }
24          for (sj = si + 1; sj < nseqs; sj++)
25          {
26             m = seqlen_array[sj+1];
27             if ( n == 0 || m == 0 ) {
28                bench_output[si*nseqs+sj] = (int) 1.0;
29             } else {
30                #pragma omp task untied \
31                private(i,gg,len2,mm_score) firstprivate(m,n,si,sj,len1) \
32                shared(nseqs, bench_output,seqlen_array,seq_array,gap_pos1,gap_pos2,
33                       pw_ge_penalty,pw_go_penalty,mat_avscore)
34                {
35                   int se1, se2, sb1, sb2, maxscore, seq1, seq2, g, gh;
36                   int displ[2*MAX_ALN_LENGTH+1];
37                   int print_ptr, last_print;
38
39                   for (i = 1, len2 = 0; i <= m; i++) {
40                      char c = seq_array[sj+1][i];
41                      if ((c != gap_pos1) && (c != gap_pos2)) len2++;
42                   }
43                   if ( dnaFlag == TRUE ) {
44                      g  = (int) ( 2 * INT_SCALE * pw_go_penalty * gap_open_scale ); // gapOpen
45                      gh = (int) (INT_SCALE * pw_ge_penalty * gap_extend_scale); //gapExtend
46                   } else {
47                      gg = pw_go_penalty + log((double) MIN(n, m)); // temporary value
48                      g  = (int) ((mat_avscore <= 0) ? (2 * INT_SCALE * gg)
49                                 : (2 * mat_avscore * gg * gap_open_scale) ); // gapOpen
50                      gh = (int) (INT_SCALE * pw_ge_penalty); //gapExtend
51                   }
52
53                   seq1 = si + 1;
54                   seq2 = sj + 1;
55
56                   forward_pass(&seq_array[seq1][0], &seq_array[seq2][0],
57                                n, m, &se1, &se2, &maxscore, g, gh);
58                   reverse_pass(&seq_array[seq1][0], &seq_array[seq2][0],
59                                se1, se2, &sb1, &sb2, maxscore, g, gh);
60
61                   print_ptr  = 1;
```

```
62                  last_print = 0;
63
64                  diff(sb1-1, sb2-1, se1-sb1+1, se2-sb2+1, 0, 0, &print_ptr,
65                       &last_print, displ, seq1, seq2, g, gh);
66                  mm_score = tracepath(sb1, sb2, &print_ptr, displ, seq1, seq2);
67
68                  if (len1 == 0 || len2 == 0) mm_score  = 0.0;
69                  else                        mm_score /= (double) MIN(len1,len2);
70
71                  bench_output[si*nseqs+sj] = (int) mm_score;
72               } // end task
73            } // end if (n == 0 || m == 0)
74         } // for (j)
75      } // end parallel for (i)
76   } // end parallel
77   bots_message(" completed!\n");
78   return 0;
79 }
```

Bibliography

[1] Herb Sutter. The Free Lunch Is Over — A Fundamental Turn Toward Concurrency in Software, original article from 2005, numbers updated 2009. URL `http://www.gotw.ca/publications/concurrency-ddj.htm`.

[2] Gordon E. Moore. Cramming More Components onto Integrated Circuits, Reprinted from Electronics, Volume 38, Number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(5):33–35, Sept 2006.

[3] Karl Rupp. 40 Years of Microprocessor Trend Data, 2015. URL `https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/`.

[4] Constantin Christmann, Erik Hebisch, and Oliver Strauß. Einsatzszenarien für Multicore-Technologien (Applikations- und Potentialanalyse) (in German), 2011. URL `http://www.mware.fraunhofer.de/`.

[5] Prakash Prabhu, Thomas B. Jablin, Arun Raman, Yun Zhang, Jialu Huang, Hanjun Kim, Nick P. Johnson, Feng Liu, Soumyadeep Ghosh, Stephen Beard, Taewook Oh, Matthew Zoufaly, David Walker, and David I. August. A Survey of the Practice of Computational Science. In *State of the Practice Reports*, SC '11, pages 19:1–19:12, New York, NY, USA, 2011. ACM.

[6] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry Standard API for Shared-memory Programming. *Computational Science & Engineering, IEEE*, 5(1): 46–55, 1998.

[7] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.5, November 2015. URL `http://www.openmp.org/mp-documents/openmp-4.5.pdf`.

[8] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[9] Ron Cytron. DOACROSS: Beyond Vectorization for Multiprocessors. In *ICPP*, 1986.

[10] Jialu Huang, Prakash Prabhu, Thomas B. Jablin, Soumyadeep Ghosh, Sotiris Apostolakis, Jae W. Lee, and David I. August. Speculatively Exploiting Cross-Invocation Parallelism. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 207–221, New York, NY, USA, 2016. ACM.

[11] Murphy Niall, Timothy Jones, Simone Campanoni, and Robert Mullins. Limits of Static Dependence Analysis for Automatic Parallelization. In *18th Workshop on Compilers for Parallel Computing (CPC)*, 2015.

[12] Anasua Bhowmik and Manoj Franklin. A General Compiler Framework for Speculative Multithreaded Processors. *IEEE Trans. Parallel Distrib. Syst.*, 15(8):713–724, August 2004.

[13] Hwansoo Han and Chau-Wen Tseng. Improving Compiler and Run-Time Support for Irregular Reductions Using Local Writes. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '98, pages 181–196, London, UK, UK, 1999. Springer-Verlag.

[14] Clemens Hammacher, Kevin Streit, Andreas Zeller, and Sebastian Hack. Thread-level Speculation with Kernel Support. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 1–11, New York, NY, USA, 2016. ACM.

[15] Kevin Streit, Johannes Doerfert, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. Generalized Task Parallelism. *ACM Trans. Archit. Code Optim.*, 12(1):8:1–8:25, April 2015.

[16] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. Polly's Polyhedral Scheduling in the Presence of Reductions. In *International Workshop on Polyhedral Compilation Techniques*, Amsterdam, Netherlands, Jan 2015.

[17] Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. Sambamba: Runtime Adaptive Parallel Execution. In *Proceedings of the 3rd International*

Workshop on Adaptive Self-Tuning Computing Systems, ADAPT '13, pages 7:1–7:6, New York, NY, USA, 2013. ACM.

[18] Johannes Doerfert, Clemens Hammacher, Kevin Streit, and Sebastian Hack. SPolly: Speculative Optimizations in the Polyhedral Model. In *International Workshop on Polyhedral Compilation Techniques*, Berlin, Germany, January 2013.

[19] Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. Sambamba: A Runtime System for Online Adaptive Parallelization. In *CC*, pages 240–243, 2012. URL `http://www.sambamba.org`.

[20] Clemens Hammacher, Kevin Streit, Sebastian Hack, and Andreas Zeller. Profiling Java Programs for Parallelism. In *Proc. 2nd International Workshop on Multi-Core Software Engineering (IWMSE)*, pages 49–55, May 2009.

[21] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[22] Frances E. Allen. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[23] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 278–289, New York, NY, USA, 2007. ACM.

[24] Katherine A. Yelick. Programming Models for Irregular Applications. *SIGPLAN Not.*, 28(1):28–31, January 1993.

[25] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem. I. One-dimensional Time. *International Journal of Parallel Programming*, 21(5):313–347, 1992.

[26] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem. part II. Multidimensional Time. *International Journal of Parallel Programming*, 21(6): 389–420, 1992.

[27] Christian Lengauer. Loop Parallelization in the Polytope Model. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, pages 398–416, 1993.

[28] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[29] Clemens Hammacher. *Efficient Runtime Systems for Speculative Parallelization.* PhD thesis, Saarland University, Faculty of Mathematics and Computer Science, 03 2017.

[30] Samuel P. Midkiff. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2012.

[31] Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. Speculative Separation for Privatization and Reductions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 359–370, New York, NY, USA, 2012. ACM.

[32] Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 218–232, New York, NY, USA, 1995. ACM.

[33] Francis H. Dang, Hao Yu, and Lawrence Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 318–, Washington, DC, USA, 2002. IEEE Computer Society.

[34] Lawrence Rauchwerger and David Padua. The Privatizing DOALL Test: A Run-time Technique for DOALL Loop Identification and Array Privatization. In *Proceedings of the 8th International Conference on Supercomputing*, ICS '94, pages 33–43, New York, NY, USA, 1994. ACM.

[35] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. A Scalable Method for Run-time Loop Parallelization. *Int. J. Parallel Program.*, 23(6):537–576, December 1995.

[36] Hao Yu and L. Rauchwerger. An Adaptive Algorithm Selection Framework for Reduction Parallelization. *Parallel and Distributed Systems, IEEE Transactions on*, 17(10):1084–1096, Oct 2006.

[37] Philip Ginsbach and Michael F. P. O'Boyle. Discovery and Exploitation of General Reductions: A Constraint Based Approach. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 269–280, Piscataway, NJ, USA, 2017. IEEE Press.

[38] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996. ISBN 1-56592-115-1.

[39] Scott Oaks and Henry Wong. *Java Threads, Third Edition.* O'Reilly Media, Inc., 3 edition, 2004. ISBN 978-0-596-00782-9.

[40] Intel ®. Threading Building Blocks (Intel ®TBB), 2013. URL `http://threadingbuildingblocks.org`.

[41] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[42] Intel ®. Cilk^{TM}Plus 1.2, 2013. URL `https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm`.

[43] Joe Armstrong. Making Reliable Distributed Systems in the Presence of Software Errors, 2003.

[44] Joe Armstrong. Erlang Programming Language, 2003. URL `http://www.erlang.org/`.

[45] Robert Griesemer, Rob Pike, and Ken Thompson. The Go Programming Language, 2009. URL `https://golang.org`.

[46] Martin Odersky. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

[47] Martin Odersky. Scala - Object-Oriented Meets Functional, 2004. URL `https://www.scala-lang.org/`.

[48] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[49] J. Mak and A. Mycroft. Critical-Path-Guided Interactive Parallelisation. In *Parallel Processing Workshops (ICPPW), 2011 40th International Conference on*, pages 427–436, Sept 2011.

[50] Jonathan Mak, Karl-Filip Faxén, Sverker Janson, and Alan Mycroft. Estimating and Exploiting Potential Parallelism by Source-level Dependence Profiling. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I*, EuroPar'10, pages 26–37, Berlin, Heidelberg, 2010. Springer-Verlag.

[51] Karl-Filip Faxén, Konstantin Popov, Sverker Jansson, and Lars Albertsson. Embla - Data Dependence Profiling for Parallel Programming. In *Proceedings of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS '08, pages 780–785, Washington, DC, USA, 2008. IEEE Computer Society.

[52] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How Much Parallelism is There in Irregular Applications? In *Proceedings*

of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '09, pages 3–14, New York, NY, USA, 2009. ACM.

[53] Michael Burke, Ron Cytron, Jeanne Ferrante, and Wilson Hsieh. Automatic Generation of Nested, Fork-join Parallelism. *The Journal of Supercomputing*, 3(2):71–88, 1989.

[54] V. Sarkar. Automatic Partitioning of a Program Dependence Graph into Parallel Tasks. *IBM J. Res. Dev.*, 35(5-6):779–804, September 1991.

[55] Daniel Cordes, Peter Marwedel, and Arindam Mallik. Automatic Parallelization of Embedded Software Using Hierarchical Task Graphs and Integer Linear Programming. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 267–276, New York, NY, USA, 2010. ACM.

[56] Radu Rugina and Martin Rinard. Automatic Parallelization of Divide and Conquer Algorithms. *SIGPLAN Not.*, 34(8):72–83, May 1999.

[57] Hongtao Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering Hidden Loop Level Parallelism in Sequential Applications. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 290–301, Feb 2008.

[58] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing Sequential Applications on Commodity Hardware Using a Low-cost Software Transactional Memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 166–176, New York, NY, USA, 2009. ACM.

[59] Hanjun Kim, Nick P. Johnson, Jae W. Lee, Scott A. Mahlke, and David I. August. Automatic Speculative DOALL for Clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 94–103, New York, NY, USA, 2012. ACM.

[60] Carlos Madriles, Pedro Lopez, Josep Maria Codina, Enric Gibert, Fernando Latorre, Alejandro Martinez, Raul Martinez, and Antonio Gonzalez. Anaphase: A Fine-Grain

Thread Decomposition Scheme for Speculative Multithreading. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 15–25, Washington, DC, USA, 2009. IEEE Computer Society.

[61] Martin Suesskraut, Thomas Knauth, Stefan Weigert, Ute Schiffel, Martin Meinhold, and Christof Fetzer. Prospect: A Compiler Framework for Speculative Parallelization. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 131–140, New York, NY, USA, 2010. ACM.

[62] Craig Zilles and Gurindar Sohi. Master/Slave Speculative Parallelization. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, pages 85–96, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[63] Paul Feautrier. Automatic Parallelization in the Polytope Model. In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, pages 79–103, 1996.

[64] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[65] Martin Griebl, Paul Feautrier, and Christian Lengauer. On Index Set Splitting. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, Newport Beach, California, USA, October 12-16, 1999*, pages 274–282, 1999.

[66] Martin Griebl, Paul Feautrier, and Christian Lengauer. Index Set Splitting. *International Journal of Parallel Programming*, 28(6):607–631, 2000.

[67] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The Polyhedral Model is More Widely Applicable Than You Think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of*

Software, International Conference on Compiler Construction, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[68] Riyadh Baghdadi, Albert Cohen, Cédric Bastoul, Louis-Noël Pouchet, and Lawrence Rauchwerger. The Potential of Synergistic Static, Dynamic and Speculative Loop Nest Optimizations for Automatic Parallelization. In Wei Liu, Scott Mahlke, and Tin fook Ngai, editors, *Pespma 2010 - Workshop on Parallel Execution of Sequential Programs on Multi-core Architecture*, Saint Malo, France, June 2010. URL `https://hal.inria.fr/inria-00494305`.

[69] Alexandra Jimborean, Philippe Clauss, Benoît Pradelle, Luis Mastrangelo, and Vincent Loechner. Adapting the Polyhedral Model as a Framework for Efficient Speculative Parallelization. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 295–296, 2012.

[70] Alexandra Jimborean, Philippe Clauss, Jean-François Dollinger, Vincent Loechner, and Juan Manuel Martinez Caamaño. Dynamic and Speculative Polyhedral Parallelization Using Compiler-Generated Skeletons. *International Journal of Parallel Programming*, 42(4):529–545, 2014.

[71] Johannes Doerfert, Tobias Grosser, and Sebastian Hack. Optimistic Loop Optimization. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 292–304, Piscataway, NJ, USA, 2017. IEEE Press.

[72] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled Software Pipelining with the Synchronization Array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 177–188, Washington, DC, USA, 2004. IEEE Computer Society.

[73] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[74] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage Decoupled Software Pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 114–123, New York, NY, USA, 2008. ACM.

[75] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative Decoupled Software Pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[76] David I. August, Jialu Huang, Stephen R. Beard, Nick P. Johnson, and Thomas B. Jablin. Automatically Exploiting Cross-invocation Parallelism Using Runtime Information. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[77] Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. Decoupled Software Pipelining Creates Parallelization Opportunities. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 121–130, New York, NY, USA, 2010. ACM.

[78] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. Parcae: A System for Flexible Parallel Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 133–144, New York, NY, USA, 2012. ACM.

[79] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. The Paralax Infrastructure: Automatic Parallelization with a Helping Hand. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 389–400, New York, NY, USA, 2010. ACM.

[80] Hans Vandierendonck and Koen De Bosschere. Automatic Parallelization in the Paralax Compiler. In *SCOPES '11 : Proceedings of the 14th International Workshop*

on Software and Compilers for Embedded Systems, pages 56–63. Association for Computing Machinery (ACM), 2011.

[81] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 84–93, New York, NY, USA, 2012. ACM.

[82] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. HELIX-RC: An Architecture-compiler Co-design for Automatic Parallelization of Irregular Programs. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, pages 217–228, Piscataway, NJ, USA, 2014. IEEE Press.

[83] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic Parallelism Requires Abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.

[84] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM.

[85] James Christopher Jenista, Yong hun Eom, and Brian Charles Demsky. OoOJava: Software Out-of-order Execution. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 57–68, New York, NY, USA, 2011. ACM.

[86] Yong hun Eom, Stephen Yang, James C. Jenista, and Brian Demsky. DOJ: Dynamically Parallelizing Object-oriented Programs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 85–96, New York, NY, USA, 2012. ACM.

[87] Michael K. Chen and Kunle Olukotun. The Jrpm System for Dynamically Parallelizing Java Programs. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 434–446, New York, NY, USA, 2003. ACM.

[88] Matthew DeVuyst, Dean M. Tullsen, and Seon Wook Kim. Runtime Parallelization of Legacy Code on a Transactional Memory System. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 127–136, New York, NY, USA, 2011. ACM.

[89] Ben Hertzberg and Kunle Olukotun. Runtime Automatic Speculative Parallelization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 64–73, Washington, DC, USA, 2011. IEEE Computer Society.

[90] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Speculative Thread Decomposition Through Empirical Optimization. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 205–214, New York, NY, USA, 2007. ACM.

[91] Benoit Pradelle, Alain Ketterlin, and Philippe Clauss. Polyhedral Parallelization of Binary Code. *ACM Trans. Archit. Code Optim.*, 8(4):39:1–39:21, January 2012.

[92] Thomas Karcher and Victor Pankratius. Run-time Automatic Performance Tuning for Multicore Applications. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6852 of *Lecture Notes in Computer Science*, pages 3–14. Springer Berlin Heidelberg, 2011.

[93] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. HELIX-UP: Relaxing Program Semantics to Unleash Parallelization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 235–245, Washington, DC, USA, 2015. IEEE Computer Society.

[94] Niall Murphy, Timothy Jones, Robert Mullins, and Simone Campanoni. Performance Implications of Transient Loop-carried Data Dependences in Automatically Parallelized Loops. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 23–33, New York, NY, USA, 2016. ACM.

[95] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[96] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 249–265, New York, NY, USA, 2017. ACM.

[97] William Moses, Tao Schardl, and Charles Leiserson. Embedding Fork-Join Parallelism into LLVM IR. In *19th Workshop on Compilers for Parallel Computing (CPC)*, 2016.

[98] Bjarne Steensgaard. Sequentializing Program Dependence Graphs for Irreducible Programs. Technical report, Microsoft Research, October 1993.

[99] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic Performance Tuning of Word-based Software Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM.

[100] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-Based Software Transactional Memory. *Parallel and Distributed Systems, IEEE Transactions on*, 21(12):1793–1807, Dec 2010.

[101] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[102] Intel ®. Transactional Synchronization in Haswell, 2012. URL `https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`.

[103] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly — Performing Polyhedral Optimizations on a Low-level Intermediate Representations. *Parallel Processing Letters*, 22(04):1250010, 2012.

[104] Jeanne Ferrante and Mary Mace. On Linearizing Parallel Code. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, pages 179–190, New York, NY, USA, 1985. ACM.

[105] J. Ferrante, M. Mace, and B. Simons. Generating Sequential Code from Parallel Code. In *Proceedings of the 2Nd International Conference on Supercomputing*, ICS '88, pages 582–592, New York, NY, USA, 1988. ACM.

[106] B. Simons, D. Alpern, and J. Ferrante. A Foundation for Sequentializing Parallel Code. In *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '90, pages 350–359, New York, NY, USA, 1990. ACM.

[107] Thomas Ball and Susan Horwitz. Constructing Control Flow from Control Dependence. Technical report, University of Wisconsin — Madison, 1992.

[108] Jia Zeng, Cristian Soviani, and Stephen A. Edwards. Generating Fast Code from Concurrent Program Dependence Graphs. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '04, pages 175–181, New York, NY, USA, 2004. ACM.

[109] Peng Tu and David A. Padua. Automatic Array Privatization. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 500–521, London, UK, UK, 1994. Springer-Verlag.

[110] Donald E. Knuth and Francis R. Stevenson. Optimal Measurement Points for Program Frequency Counts. *BIT Numerical Mathematics*, 13(3):313–322, 1973.

[111] Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, July 1994. ISSN 0164-0925.

[112] T.J.K. Edler von Koch and B. Franke. Variability of Data Dependences and Control Flow. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 180–189, March 2014.

[113] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying Behavioral Differences Between C and C++ Programs. *Journal of Programming Languages*, 2:313–351, 1995.

[114] Andrew Tridgell and Joel Rosdahl. ccache — A Fast C/C++ Compiler Cache, 2016. URL `https://ccache.samba.org/`.

[115] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[116] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[117] Louis-Noël Pouchet. PolyBench/C: The Polyhedral Benchmark Suite, 2012. URL `http://www.cs.ucla.edu/~pouchet/software/polybench/`.

[118] Sid Ahmed Ali Touati, Julien Worms, and Sébastien Briais. The Speedup-Test: a Statistical Methodology for Programme Speedup Analysis and Computation. *Concurrency and Computation: Practice and Experience*, 25(10):1410–1426, 2013.

[119] Alan A.A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 1st edition, 2015. ISBN 0134190440, 9780134190440.

[120] Nicholas D. Matsakis and Felix S. Klock, II. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, 2014. ACM.

[121] The Rust Community. The Rust Language (Website), 2014. URL `https://www.rust-lang.org`.

[122] Ben Hardekopf and Calvin Lin. Flow-sensitive Pointer Analysis for Millions of Lines of Code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 289–298, Washington, DC, USA, 2011. IEEE Computer Society.

[123] Samuel Z. Guyer and Calvin Lin. Error Checking with Client-driven Pointer Analysis. *Sci. Comput. Program.*, 58(1-2):83–114, October 2005.

[124] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid Analysis: Static & Dynamic Memory Reference Analysis. *Int. J. Parallel Program.*, 31(4):251–283, August 2003.

Multi core systems are ubiquitous nowadays and their number is ever increasing. And while, limited by physical constraints, the computational power of the individual cores has been stagnating or even declining for years, a solution to effectively utilize the computational power that comes with the additional cores is yet to be found.

Existing approaches to automatic parallelization are often highly specialized to exploit the parallelism of specific program patterns, and thus to parallelize a small subset of programs only. In addition, frequently used invasive runtime systems prohibit the combination of different approaches, which impedes the practicality of automatic parallelization.

In this thesis, we show that specializing to narrowly defined program patterns is not necessary to efficiently parallelize applications coming from different domains. We develop a generalizing approach to parallelization, which, driven by an underlying mathematical optimization problem, is able to make qualified parallelization decisions taking into account the involved runtime overhead. In combination with a specializing, adaptive runtime system the approach is able to match and even exceed the performance results achieved by specialized approaches.